ARO 24753.2-EL

A Final Report
Covering the Period
July 1, 1989 - December 31, 1989

PHOENIX, A HIGH-PERFORMANCE UNIX WITH AN EMPHASIS ON DYNAMIC MODIFICATION, REAL-TIME RESPONSE AND SURVIVABILITY

Contract No. DAAL03-87-K-0090

Project Period: June 1, 1987 - September 30, 1989

Submitted to:

U. S. Army Research Office
P. O. Box 12211
4300 S. Miami Boulevard
Research Triangle Park, NC 27709

Submitted by:

R. P. Cook
Associate Professor

Report No. UVA/525186/CS90/102
January 1990

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF
ENGINEERING
& APPLIED SCIENCE

University of Virginia
Thornton Hall
Charlottesville, VA 22903

UNIVERSITY OF VIRGINIA

School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 600. There are 160 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Applied Mechanics, Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics, Applied Mathematics and Computer Science. Within these disciplines there are well equipped laboratories for conducting highly specialized research. All departments offer the doctorate; Biomedical and Materials Science grant only graduate degrees. In addition, courses in the humanities are offered within the School.

The University of Virginia (which includes approximately 2,000 faculty and a total of full-time student enrollment of about 17,000), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.

A Final Report
Covering the Period
July 1, 1989 - December 31, 1989

PHOENIX, A HIGH-PERFORMANCE UNIX WITH AN EMPHASIS ON DYNAMIC MODIFICATION, REAL-TIME RESPONSE AND SURVIVABILITY

Contract No. DAAL03-87-K-0090

Project Period: June 1, 1987 - September 30, 1989

Submitted to:

U. S. Army Research Office
P. O. Box 12211
4300 S. Miami Boulevard
Research Triangle Park, NC 27709

Submitted by:

R. P. Cook
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/525186/CS90/102
January 1990

Copy No. 47

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UVA/525186/CS90/102	5. MONITORING ORGANIZATION REPORT NUMBER(S) ARO 24753.2-EL
6a. NAME OF PERFORMING ORGANIZATION University of Virginia Dept. of Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research Resident Representative
6c. ADDRESS (City, State, and ZIP Code) Thornton Hall Charlottesville, VA 22903-2442	7b. ADDRESS (City, State, and ZIP Code) 2135 Wisconsin Avenue, N.W. Suite 102 Washington, DC 20007
8a. NAME OF FUNDING / SPONSORING ORGANIZATION U. S. Army Research Office 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAL03-87-K-0090
8c. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
PHOENIX, A High-Performance UNIX With an Emphasis on Dynamic Modification, Real-Time Response and Survivability

12. PERSONAL AUTHOR(S)
R. P. Cook

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 7/1/89 TO 12/31/89	90 January 12	64

16. SUPPLEMENTARY NOTATION
The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Operating Systems, Real-Time

19. ABSTRACT (

The goal of the Phoenix research project, which is a three year effort, is to develop a high-performance operating system for embedded applications that have a real-time response requirement. The system is to be extremely modular so that it can be easily adapted to meet different performance goals or application restrictions. Phoenix will also support a UNIX-like system call interface for compatibility with government standards. There are currently no UNIX operating systems capable of meeting "hard" real-time requirements. There are currently no UNIX operating systems that can be easily adapted to meet application requirements.

We will also investigate the problems associated with modifying an operating system and application programs remotely without halting the system. For real-time systems, the modifications must be performed in such a way that the unavailability of the system, or particular modules, is minimized.

Another aspect of the project is the analysis of operating system construction techniques that minimize the unavailability of the system when a power failure or hardware malfunction occurs and that maximize the ability of a system to "pick up" where it left off. Other areas of investigation include operating system structuring techniques, better algorithms, and better system interfaces.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

TABLE OF CONTENTS

FINAL PROJECT REPORT

1. ARO PROPOSAL NUMBER: 24753-EL

2. PERIOD COVERED BY REPORT: 1 July 1989 - 31 December 1989

3. TITLE OF PROPOSAL: PHOENIX, A High-Performamce UNIX With an Emphasis on Dynamic Modification, Real-time Response & Survivability

4. CONTRACT OR GRANT NUMBER: DAAL03-87-K-0090

5. NAME OF INSTITUTION: University of Virginia

6. AUTHORS OF REPORT: Robert P. Cook

7. LIST OF MANUSCRIPTS SUBMITTED OR PUBLISHED UNDER ARO SPONSORSHIP DURING THIS REPORTING PERIOD, INCLUDING JOURNAL REFERENCES:

(published) The StarLite Operating System in the **1989 Workshop on Operating Systems for Mission Critical Computing**, (Sept.1989).

(to appear) StarLite -- A Software Education Laboratory in the **4th SEI Conference on Software Engineering Education**, (April 1990).

(submitted) StarLite -- A Laboratory for Operating Systems Research to the **Journal of Parallel and Distributed Computing, Special Issue on Software Tools.**

(submitted) The StarLite Programming Environment to the **ACM SIGPLAN 90 Conference on Programming Language Design and Implementation.**

(submitted) Efficient Locking for Process Control Operations to the **Computing Systems Journal.**

(submitted) RDB: An Open, Real-Time, Relational Database Kernel to the **EUROMICRO 1990 Conference.**

R. P. Cook
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

8. SCIENTIFIC PERSONNEL SUPPORTED BY THIS PROJECT AND DEGREES AWARDED DURING THIS REPORTING PERIOD:

Robert P. Cook, PI
Chris Koeritz, M.S. student
Richard McDaniel, B.S. student
Ambar Sarkar, Ph.D. student

Outside technical activities pertinent to the contract included participation in the SEI and Aerospace Industries Association workshop on "Research Advances Required for Real-Time Software Systems in the 90's". I was one of the few invited participants from academia.

I accepted the position as Chairman of the Seventh IEEE Workshop on Real-Time Operating Systems and Software, which will be held in Charlottesville on May 10, 1990. I will also be participating in the IEEE P1151 standards effort for Modula-2. I also participated in a workshop held at the Virginia Center for Innovative Technology for state researchers in the real-time area. Furthermore, the Department has made it to the site visit stage in this year's NSF CISE competition. The Phoenix operating system, running in the StarLite environment, was one of the two demonstrations that the department chose to impress the visitors. They were very impressed.

Of the ten new Ph.D. students in this year's entering graduate class, five are working with me on the Phoenix project. The group has the best background and intellect of any of the students that I have worked with at Virginia. I have also been fortunate to attract a top-rank minority Ph.D student. These students, if properly supported, will have a dramatic impact on my research.

One of the students is performing an empirical analysis of novel resource allocation algorithms for real-time operating systems. Another is implementing a data-flow, protocol engine that is an improvement on Larry Peterson's work at Arizona on high-performance protocol implementations. Another is designing and implementing fault-tolerant multicast protocols for the distributed version of Phoenix. Another is running experiments on real-time scheduling algorithms. Finally, another student is experimenting with optimal locking schemes for multiprocessor operating system implementations.

For the real-time prototyping environment, we implemented a dynamic filter attachment option for the debugger so that data could be displayed and manipulated through application-dependent views. We are also working on a translator that will produce native-code versions of our library's object modules. Once the translator is complete, the interpreter will be modified to load native-code modules if they are present. Measurements indicate that we should be able to run at close to machine speeds while maintaining compatibility with the environment's tool set.

R. P. Cook
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

MANUSCRIPTS

The StarLite Operating System

Robert P. Cook*
cook@cs.virginia.edu
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
(804) 979-9943

1.0 Introduction

The StarLite project [1,2,3] has four research components in the areas of prototyping, operating systems, database, and computer network technology. The prototyping environment, which executes on Sun workstations, supports the development and execution of software for uni- or multi-processors, as well as distributed systems.

Figure 1 illustrates the use of the prototyping environment during a test session for the StarLite operating system. The figure illustrates our proprietary UNIX* implementation "booting up" on a six-node virtual network. Once the virtual network has booted, the system designer can execute test programs, collect statistics, or examine the system state using the builtin debugger, which is illustrated in Figure 2. We have invested a good deal of effort in building the prototyping system to create what we feel is the best possible research environment.

The purpose of this paper is to describe our approach to designing a new operating system for mission-critical computing and to review some of the technology issues being explored as part of the StarLite project.

2.0 Operating System Interfaces

In this Section, we describe the interface requirements that we feel would be most appropriate for a mission-critical operating system solution. Interfaces are important because they can be standardized and because they are designed to outlive implementations and machine architectures.

It is now widely accepted that the use of a procedural interface, such as the C library for UNIX, is the most advantageous method for presenting an operating system's functionality to an end user. Such an interface can be machine and language invariant. These are desirable properties given the diversity of hardware/software used by today's defense contractors.

There are two design options to choose from as the basis for an interface standard: **flat** and **layered**. An operating system with a flat interface, such as UNIX, is essentially closed; that is none of the interfaces used in the implementation can be accessed. Flat interfaces are inflexible and typically trade performance and control for generality.

A layered interface specification, such as the ISO OSI definition for computer networks, overcomes the deficiencies of the traditional, flat

*This work is supported by by ONR under contract N00014-86-K0245 and ARO under contract DAAL03-87-K0090.
*UNIX is a registered trademark of AT&T Bell Laboratories.

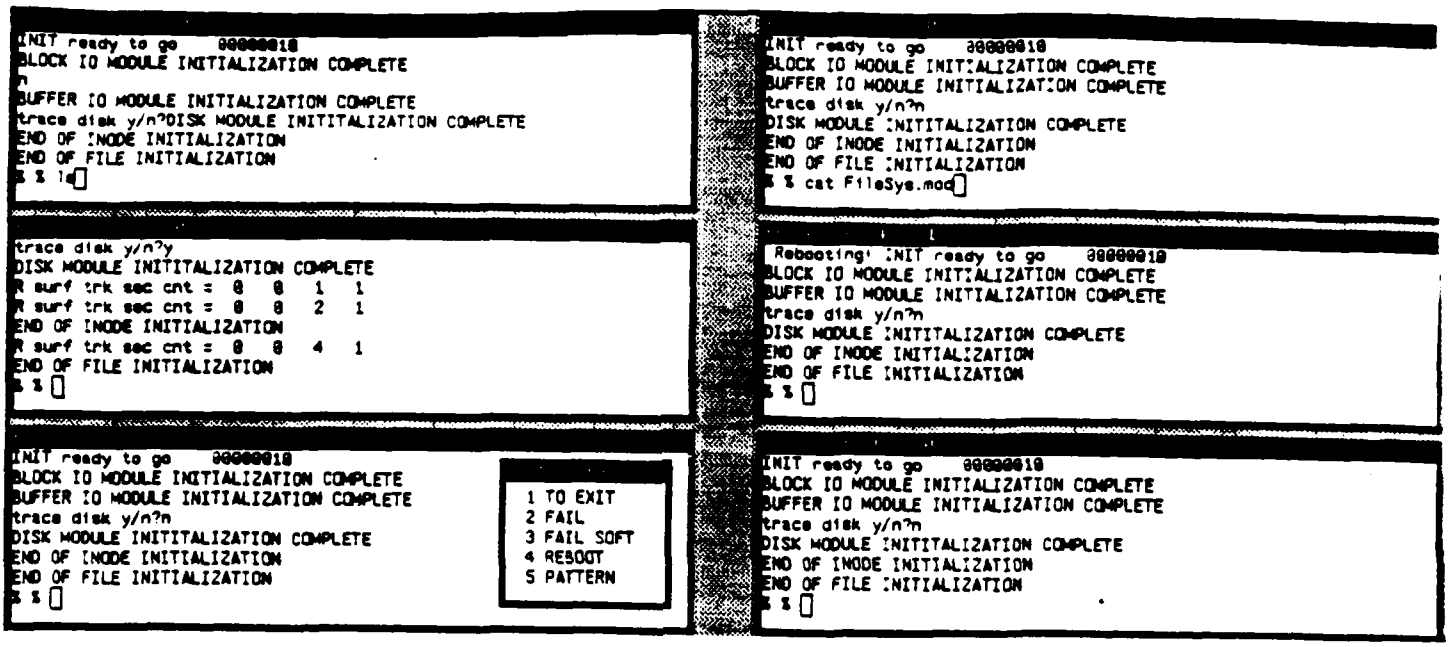

Figure 1. A Six-Node StarLite System

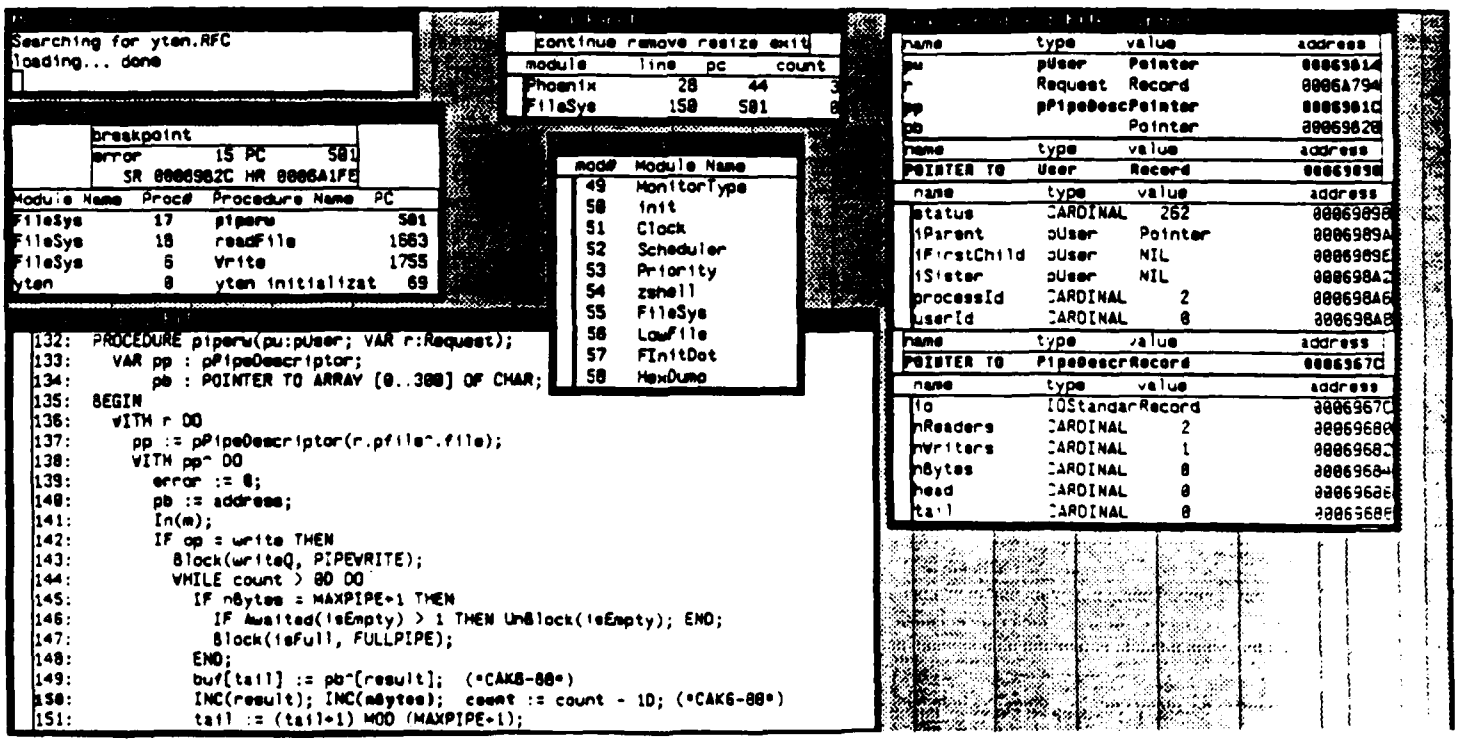

Figure 2. The StarLite Symbolic Debugger

operating system interface designs by allowing the application engineer to choose an interface layer that most closely fits the problem to be solved. For example, if UNIX were a layered design, it would be possible for a database system to manipulate the operating system's buffer cache in a manner that has long been requested by implementors[4].

Access to low-level interfaces can address the performance requirements of mission-critical software. Another advantage of a layered design is that layers can be omitted to save space. For example, if an application does not use files, the file system could be omitted. It is also possible to implement layers in hardware to improve performance.

The StarLite operating system is based on a layered design with standard interfaces. Two of the research issues are how to partition the layers and how to define the interfaces at each layer.

To experiment with different options, we designed and implemented a UNIX-compatible operating system according to the layering principles defined by ISO[5]. The StarLite UNIX is proprietary in that it is not based upon nor does it contain any code from other UNIX implementations. We have rewritten the system several times to try different layering and implementation strategies.

We have found interface specification to be a more demanding task than doing the implementation. In other words, writing a monolithic piece of code to solve a problem is much easier than creating a layered design in which the layers are intended to form functionally complete and useful subsets.

We have found two other problems with a layered design that we are addressing as part of our research. The first problem is the overhead of procedure calls through multiple layers of software. The second problem results from application requirements, such as protection features, which can affect interfaces and implementations at a number of layers. Even if the requirement is removed at a higher layer, there may be unused procedures and data structures at lower layers that affect performance. Both problems are being solved by improving compiler technology.

3.0 Interface Implementations

Most operating system implementations are **closed**; that is, the user cannot and probably should not modify them. The StarLite operating system is designed to support an arbitrary number of different, validated implementations for a given interface. As a result, the operating system as a whole follows an **open** systems architecture that can be tuned to meet application requirements. Examples of different implementation options for the same interface specification include CPU and disk scheduling algorithms or hierarchical versus flat-file name interpretation.

The long-term goal of the StarLite project is to create an operating system generator that could automatically select implementations from a module library based on specified application requirements and a given target architecture. The first step toward achieving this goal is to create a library of implementation modules suitable for mission critical applications. The current phase of the StarLite project is concerned with creating such a library.

4.0 A Software Backplane

One of the prerequisites for experimenting with a library of operating system components is having the ability to add and delete modules or services. Also, we felt that some composition mechanism would be necessary to achieve the goal of creating an operating system generator.

This section discusses the two components of the StarLite operating system that make up what we call a **software backplane**. The two components are a composition strategy for process

objects and a dynamic binding option for system services. The first component is used to create the internal structure of an operating system; the second is used to connect various services to that underlying structure.

In a traditional operating system implementation, such as UNIX, the properties of a process are stored as a single record. Any changes in one module of UNIX require a change in the ".h" file for the shared record. The result is that all the modules in the system must then be recompiled.

The StarLite composition mechanism eliminates unnecessary recompilations by binding properties to processes dynamically. The method is object based but does not support inheritance. Thus, the support code is small and fast.

In StarLite, there is only one class of object, a process. Each process object can be composed of a limited number of properties that can be connected to it in any order and at any time.

When the operating system boots up, each module has been statically linked to the code of the modules that it depends on. However, each module dynamically connects its data type to the process object using a low-level **creat** system call. For example, 'creat(">process/TimingInfo")' would append a set of timing properties of a certain size to every process object. The property fields are created only once when the system boots up. Also at boot time, the modules that use a particular property retrieve the location of its fields with an **open** system call, such as 'open(">process/TimingInfo")'. Again, this only occurs once. Note that the net effect is the same as being able to declare a RECORD structure with the field location bound at runtime.

In order to use the TimingInfo property, a module must execute a **read** system call to retrieve a pointer to or copy of the desired field, depending on the semantics. The contents of the field can then be manipulated with the same efficiency as the traditional UNIX process structure. For example, "p^.nextTimeOut" would retrieve a field from the TimingInfo record.

It is also possible to associate managers with properties. When a process object is closed, the managers are notified one at a time so that the individual fields may be closed. For example, the **exit** system call's implementation is unaware that there is a file system associated with a process. When the manager of the file-system property is invoked as the result of a process exit, it does its own cleanup by closing all open files.

Managers can also be used to monitor the actions on fields for debugging purposes. This is somewhat equivalent to the probe points used on hardware backplanes.

The second component of StarLite's software backplane design is its user services interface. The operating system acts as an agent between user interfaces and modules that provide services. The connection between the two is by means of messages in which the operations requested can be **open**, **share**, **read**, **write**, **rcontrol**, **wcontrol**, and **close**. However, the interpretation of the message is strictly up to the service modules. Thus, the system implementor can create an arbitrary number of user interfaces and an arbitrary number of implementations of those interfaces.

For example, assume that a user **opens** "/dev/pipe". The result is that an action procedure is dynamically associated with the IO field in the user's process object. Next, an open message is constructed and sent to the Pipe module. The return value, which represents two file descriptor tags for the read/write ends of the pipe, is sent to the user's process as a result. The applications engineer can choose from a variety of pipe implementations by using different names. Note that dynamic binding need not entail demand loading; the implementation modules can be loaded with the boot image if desired.

The user services interface has one other aspect, the notion of **context**, that we feel is important for mission critical computing. A context defines the mechanism by which names are interpreted. In the StarLite implementation, all name resolution is accomplished by messages sent to context services by means of action procedure calls.

As a result, any path name syntax and any effect can be realized. For example, the dynamic service binding is implemented by a context module. Contexts can also be used for performance enhancement. For instance, the standard UNIX implementation of path name resolution can result in lengthy and unpredictable disk accesses. Critical read-only file names could be resolved by a context so that their index and data blocks were locked in memory thereby achieving unit access times.

We feel that adaptability and extensibility are desirable properties for operating systems to support mission critical computing. The traditional methods of changing interfaces as new application and technological requirements arise are unacceptable. StarLite achieves flexibility without sacrificing performance.

5.0 Technology Issues

In this section, we discuss some of the StarLite project's research in operating system implementation techniques. The areas discussed include a Volume Storage Format standard (VSF), synchronization, and resource allocation.

5.1 A VSF standard

We feel that one of the key aspects of a support stategy for mission critical computing is a standard format for disk volumes. The advantages are that this standard could be implemented in hardware for high-performance and that files stored on any volume could be accessed by any operating system.

One way to achieve this goal would be to dictate the use of a standard file system for all critical computing. This may not be feasible so we have investigated the lesser goal of standardizing file manipulation, indexing, and disk space allocation. Each vendor's operating system is then presented with a standard interface to a volume.

At the current time, the **VSF** standard is designed to maintain the integrity of a volume's bit map, file descriptors and index blocks. It is up to each operating system to maintain the consistency of other information, which may be arbitrary. For example, UNIX information, such as access times or an owner's id, could be manipulated freely through the interface. Each operating system is free to add whatever information that it wants to either file descriptors or index blocks.

This flexibility is achieved by partitioning the descriptor and index blocks into two parts. One part can be manipulated arbitrarily by the host operating system through a protected interface. The second part can only be used in certain restricted, but always safe, ways. The integrity of the protected information, which contains disk block addresses, guarantees the integrity and recoverability of a volume's data.

The protected part of an index block or file descriptor contains index slots. Each index slot can identify an extent, which can be as small as one block, or another index block. For high performance applications, each file can be implemented as a single extent consisting of a file descriptor followed by the data. This organization avoids the overhead associated with the traditional UNIX implementation.

The design also supports the creation of multilevel index structures. Since an operating system can store into the unprotected part of an index block, it is possible to efficiently implement keyed access methods, such as B-trees, that do not "fit" into the UNIX filesystem model. Although we have not tried it yet, it is also possible to create indices that span multiple files.

The proposed standard is flexible, supports volume interchange, and can be used to achieve predictable, high-performance operation. Proprietary file systems can still be defined, but low-level access to data across systems is guaranteed.

5.2 Synchronization

The StarLite operating system is implemented using the hierarchy of synchronization abstractions listed in Figure 3. Operators lower in the hierarchy have higher performance but have undesirable side-effects associated with their use. Disabling interrupts to protect critical sections is fast (usually one machine instruction) but its indiscriminate use can increase interrupt latency times, which in turn can affect critical event response times. The technique is also inappropriate for multiprocessors where disabling interrupts on one processor has no effect on the execution of the others. The use of DISABLE in StarLite is restricted to two standard modules plus any device drivers that implement device synchronous operations.

As a result, StarLite minimizes interrupt latency. Furthermore, the fine granularity of locking supports kernel preemption as well as simultaneous system or IO operations.

Synchronization Operation	Level
DISABLE/RESTORE	1
Spin Locks	1
Semaphores	2
Monitors	3
Blocking	4

Figure 3. Synchronization Operations

At the higher layers of the StarLite implementation, Semaphores are used in protect critical sections that consist of straight-line code; Monitors are used for critical sections with blocking conditions; and Blocking operations are used for the cases in which a delayed thread can be swapped out. For swapped, blocked threads, the unblock operation is reflected as a state change that defers the wakeup signal until the process is swapped in and scheduled to run.

In addition to experimenting with fine-grained locking and synchronization techniques for operating system construction, we are also investigating the enhancements necessary to support real-time. Two areas of interest are priority inheritance schemes and an integrated view of criticality.

5.3 Resource allocation

Management of resources is one of the most difficult problems to solve in order to produce a full-function UNIX operating system that is capable of providing hard, real-time guarantees.

The problem occurs when a low-priority process holds a resource requested by a high-priority process. If the resource cannot be preempted or released quickly enough, the high-priority process can miss its deadline. The second part of the real-time guarantee problem is to make system timings predictable in the absence of resource contention.

The current StarLite implementation attempts to guarantee that the highest-priority process executes in an interference-free manner as long as its resources are disjoint from other processes. For example, disk writes would circumvent disk scheduling and would supercede other requests.

Our approach to the resource contention problem is based on priority-ordered avoidance[6]. This technique requires that tasks with "hard" deadlines submit claims describing future actions and timing requirements. The system then guarantees that the deadline will be met as long as the task does not exceed its computation and resource limits and neither the hardware nor

software fail.

Each process with "hard" deadlines must submit a claim list identifying the resources to be used and the timing requirements. The system then associates a data structure with each resource that restricts access by competing processes during critical periods. The key to success is making the avoidance test fast enough, which is achieved by using priority to totally order the necessary comparisons.

6.0 Summary

StarLite is a research project that is exploring new ideas for operating system structuring, interface design, analysis, and implementation. It is one of the few projects that provides a standard UNIX interface together with an implementation strategy that addresses the critical system needs of high-performance, openess, and predictability. Its layered interface design and software backplane implementation strategy make the StarLite system unique. Furthermore, its distribution as part of the StarLite programming environment means that any researcher with a SUN workstation can work with the StarLite design to make it better.

References

[1] Cook, R.P., StarLite, A Visual Simulation Package for Software Prototyping, **Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments**, (Dec. 1986) 102-110, also **SIGPLAN Notices 22**, 1(Jan. 1987).

[2] Cook, R.P., StarMod, A Language for Distributed Programming, reprinted in **Concurrent Programming**, Addison-Wesley, edited by N. Gehani and A.D. McGettrick, (1988).

[3] Son, S.H. and R.P. Cook, Scheduling and Consistency in Real-Time Database Systems, **Sixth IEEE Workshop on Real-Time Software and Operating Systems**, (May 1989) 42-45.

[4] Gray, J.N., Notes on Database Operating Systems, in **Operating Systems--An Advanced Course**, edited by Bayer, Graham, Seegmuller , (1979).

[5] Zimmermann, H., OSI Reference Model--The ISO Model of Architecture for Open Systems Interconnection, **IEEE Transactions on Communications COM-28**, (April 1980) 425-432.

[6] Munch-Anderson, B. and T.U. Zahle, Scheduling According to Job Priority With Prevention of Deadlock and Permanent Blocking, **Acta Informatica 6**, 3(1977) 153-175.

-- StarLite --
A Software Education Laboratory

Robert P. Cook
Lifeng Hsu
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

Laboratories are a prerequisite to all scientific investigation. The ability to quickly create experiments amplifies a scientist's intellectual ability. For education, students experiment to learn and to gain experience. Ideally, a laboratory helps a student visualize a problem domain, such as concurrent or distributed systems. Eventually, the students learn to reason about a problem domain abstractly, but typically the experience must come first.

1.0 Introduction

Software support for distributed programming has lagged far behind hardware development because of the complexity of the area and the lack of experimentation. In the past, to experiment with software for distributed or multiprocessor systems, a systems researcher had to purchase multiple CPUs and network interfaces. Such an investment is impractical for many people. Furthermore, even to those with network access, tools for conducting research on software systems for parallel and distributed system are non-existent. Now, with the StarLite environment, only a single computer is required.

StarLite achieves this by isolating the machine-dependent portions of a software system using modular programming techniques to provide machine-independent interfaces to simulate the behavior of the target hardware. Each device in the StarLite environment is presented to the user as an abstract data type. For instance, a Modula-2 **definition** module is used to represent the test software's view of a machine model, while a Modula-2 **implementation** module does the emulation of the characteristics of physical machines.

The StarLite environment currently supports four research areas: **programming environments, operating systems, database,** and **computer network** technology. The environment includes a Modula-2 compiler, an interpreter, a window package, a viewer, and an optional simulation package. The compiler and interpreter are implemented in C for portability. The rest of the software is in Modula-2. The system currently runs on SUN workstations and PCs.

We are currently teaching a distributed operating systems course that uses StarLite. The course is based on domain analysis and interface design as the foundation for an investigation of implementation options. It is open to upper level undergraduate and graduate students. The course load depends on each student's interest and background. The basis of study is a UNIX variant, Phoenix, that we are developing using StarLite. Since the C code for the compiler and interpreter are not relevant to the course, only the StarLite components written in Modula-2 are presented to the class.

We emphasize the following general themes throughout the course:

- Selection and utilization of appropriate tools
- Modular decomposition
- Software reuse
- Layered interfaces
- Open architecture

and the general requirements are as follows:

- Study problems of software development.
- Construct parallel and distributed programs by individual effort using StarLite's concurrent and distributed programming kernels.
- Construct a small-scale distributed operating system and use StarLite tools to debug the system and to analyze the performance of the program.
- Present the project. This includes documentation and evaluation.

2.0 Approaches and Tools

This section discusses our pedagogical approach and the tools used in the course. The descriptions are divided into the following four parts:

1. Methodology: Operating system design philosophies are described.

2. Programming and Tools: This part described the programming exercises that students do and the StarLite tools.

3. Observation: Some observations from past experiments are presented.

4. Examples: Some examples of the use of StarLite for multi-processor and distributed systems are presented.

2.1 Methodology

In addition to various topics on Operating Systems, basic problems and concepts of software development are introduced. Also, operating System design philosophies and the StarLite tools are discussed.

The students are normally given questions before class for discussion. Typical questions include: How many ways are there to implement an abstract data type? How often do the students code the same algorithms for different applications? How can object-based programming paradigms affect operating system design? These questions will eventually lead to a discussion of techniques for modular decomposition with an emphasis on software reuse. Examples are chosen from paging, resource allocation, and file systems design.

2.1.1 Modular decomposition and software reuse

The current problems in software education come from many sources. First, of the three kinds of abstractions-- procedural, control, and data abstractions-- only procedural abstraction is supported well by conventional languages[1] which, unfortunately, still dominate most of our educational environments. Secondly, our pedagogical approaches have long been focused on **problems of implementations** rather than **problems of specification**, while the later often turn out to be more fundamental in the long run. Finally, even if specification is stressed, operational specifications[2] rather than abstract specifications are still the major themes of many software design courses. Thus, the common dilemma of higher-level software courses is that the students almost always end up decomposing the **program** instead of the **problem**. Therefore, for every application and problem, we challenge the students to construct interfaces that could potentially last forever rather than writing programs just to solve the problem of the day.

2.1.2 Layered interfaces

Generally there are two design options to choose from as the basis for an interface standard: **flat** and **layered**. A system with a flat interface, such as UNIX operating system, is essentially closed; that is none of the interfaces used in the implementation can be accessed. Flat interfaces are inflexible and typically trade performance and control for generality.

A layered interface specification, such as the ISO OSI[3] definition for computer networks, overcomes the deficiencies of the traditional, flat operating system interface designs by allowing the application engineer to choose an interface layer that most closely fits the problem to be solved.

Another advantage of a layered design is that layers can be omitted to save space. For example, if an application does not use files, the file system could be omitted. It is also possible to implement layers in hardware to improve performance.

More importantly, by using the layered approach, the design and implementation of the system is simplified. As a result, clean and lucid user interfaces are easier to construct, and the system is much more amenable to investigation and experimentation.

2.1.3 Open architecture

StarLite is designed to support an arbitrary number of different, validated implementations for a given interface. As a result, the system as a whole is designed as an **open** systems architecture that can be tuned to meet application requirements. Examples of different implementation options that can be used with the same interface specification include CPU and disk scheduling algorithms or hierarchical versus flat-file name interpretation.

The long-term goal of the StarLite project is to create a system generator that could automatically select implementations from a module library based on specified application requirements and a given target architecture. The first step toward achieving this goal is to create a library of implementation modules suitable for mission critical applications. The orientation of the class is concerned with creating such a library.

2.2 Programming and Tools

We believe that the fundamental techniques used to design software for uni-processor and multi-processor machines are basically the same. We also believe that a fundamental understanding of concurrent programming concepts is essential to constructing a correct and efficient program. Each student, therefore, is required to construct some parallel and distributed programs using the StarLite concurrent and distributed programming kernels. The implementations of **coroutine** and **process** abstractions as well as various synchronization and interprocess communication models[4] are studied in great detail.

Students are also required to experiment with StarLite tools such as the Viewer, and Profiler. The StarLite Viewer subsumes the functionality of a traditional debugger, but is quite a bit more. First, the Viewer allows the user to explore, monitor and modify any thread, module, procedure, or variable on any processor. Also, all hardware details are accessible from the viewer. An example of the Viewer is given in Figure 1.

Figure 2 demonstrates the StarLite Profiler. The user may monitor the execution of the system by inspecting the frequency distribution charts. The different charts include frequency distributions of where the sytem is spending its time by module number, where selected modules are spending their time by program counter values, and op code usage in the architecture. With the aid of the Profiler, hot spots in the program can be easily detected and then the user may tune the code to improve system performance.

Finally, each student is expected to do a semester project. Since the system is **open**, a student may either evaluate alternative implementations for existing interfaces, or construct additional layers of software.

2.3 Observations

StarLite is based on a layered design with standard interfaces. Two of the research issues are how to partition the layers and how to define the interfaces at each layer.

To experiment with different options, the class designed and implemented a UNIX-compatible operating system according to the layering principles defined by ISO[3]. The StarLite UNIX is proprietary in that it is not based upon nor does it contain any code from other UNIX implementations. We, including the students in the past, have rewritten the system several times to try different layering and implementation strategies. We have found interface specification to be a more demanding task than doing the implementation. In other words, writing a monolithic piece of code to solve a problem is much easier than creating a layered design in which the layers are intended to form functionally complete and useful subsets that have a lifetime beyond the program in which they are contained.

We have also found two other problems with the layered design. The first problem is the overhead of procedure calls through multiple layers of software. The second problem results from application requirements, such as protection features, which can affect interfaces and implementations at a number of layers. Even if the requirement is removed at a higher layer, there may be unused procedures and data structures at lower layers that affect performance. Both problems are being solved by improving compiler technology.

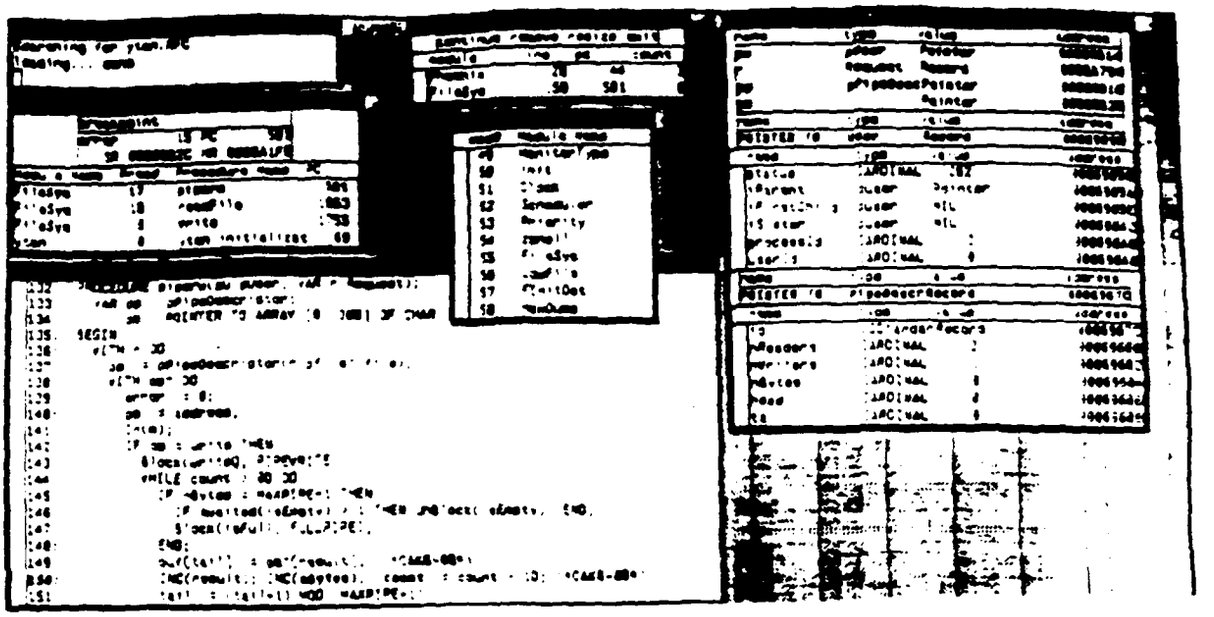

Figure 1 . The StarLite Viewer

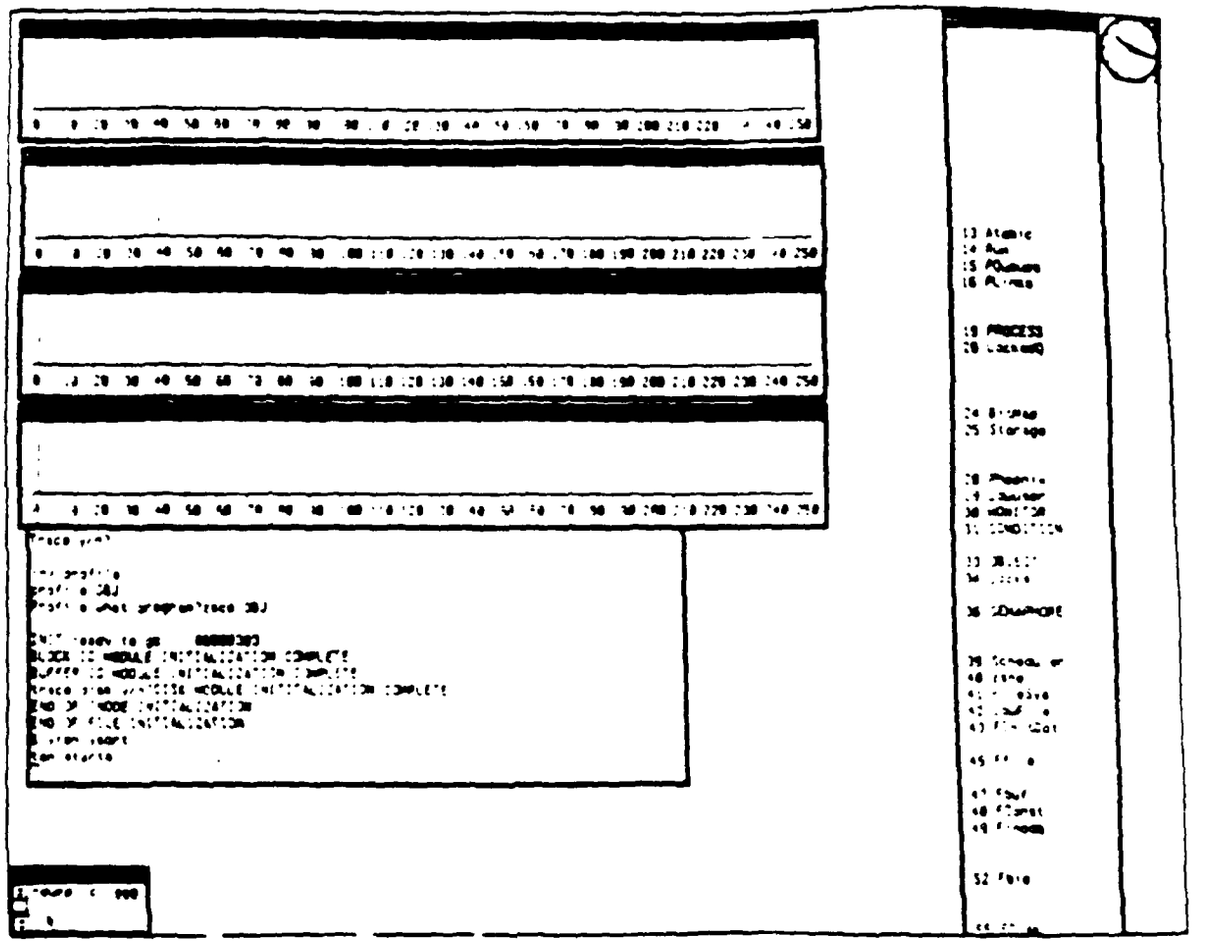

Figure 2. The StarLite Profiler

2.4 Examples:

2.4.1 StarLite Machine Models

The StarLite interpreter supports the simultaneous execution of multiple virtual processors in a single address space. Figure 3 describes the three virtual machine models supported by StarLite: single processor, multiprocessor, and distributed processors. All software developed in the laboratory uses one of these machine models as a base. For distributed processors, each virtual processor has its own copy of the test software. For the other machine models, the software is shared by all processors.

2.4.2 Multiprocessor Machine Models

Figure 4 illustrates a program running on a multiprocessor system. Each small box indicates the current state of an individual processor. The letter R stands for Running, and I for Idle. The example shows that only three processors are being utilized.

Figure 5 shows several of the StarLite visualization abstract data types. In the State Queue data type, the shaded area in each square indicates a frequency count for a single simulation entity. This visualization aid is particularly useful for spotting system bottlenecks, such as long delay lines for queues. The letter 'v' in the disk window indicates the current position of the read/write head over the surface of the disk.

2.4.3 Distributed System Models

Figure 6 illustrates the user view of a 4-node network. Each node has a window that represents its console. Furthermore, each node is running the UNIX variant developed by the class. Two of the nodes have booted up to the shell level and are ready to accept user commands. One of the nodes has tracing enabled for its disk actions. After the nodes have booted, the StarLite user can execute system tests, collect statistics, or examine/modify the system state.

Figure 7 presents parts of the environment for testing network protocols. The student first indicates the transmitting and receiving parties. When the execution is invoked, each window displays the detailed activities of each node. The student may also test the protocols under different failure assumptions. For example, how would the protocol perform if one of the nodes temporarily fails? By using the FAIL and REBOOT options, the student may crash and reboot nodes and then observe the network performance or fault-tolerance of their algorithms.

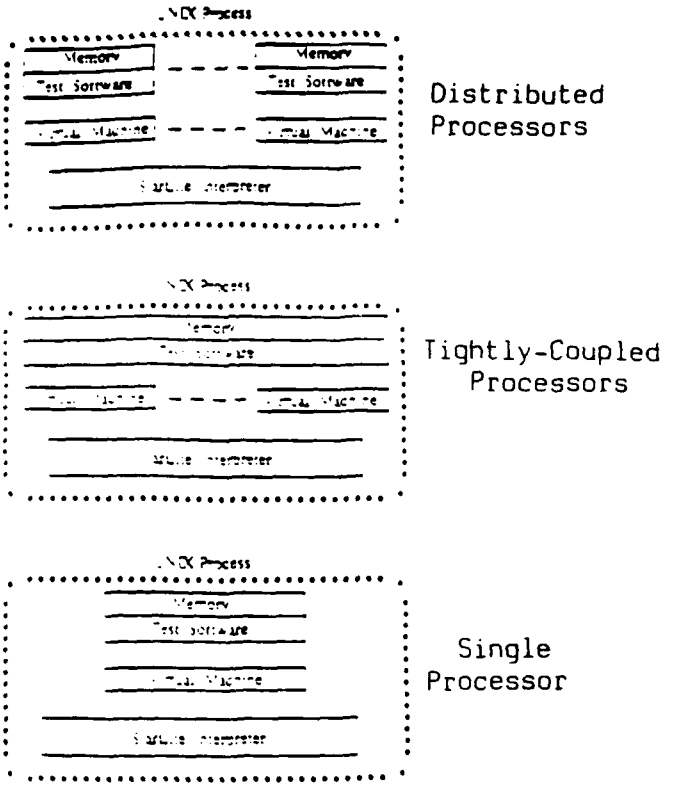

Figure 3. StarLite Machine Models

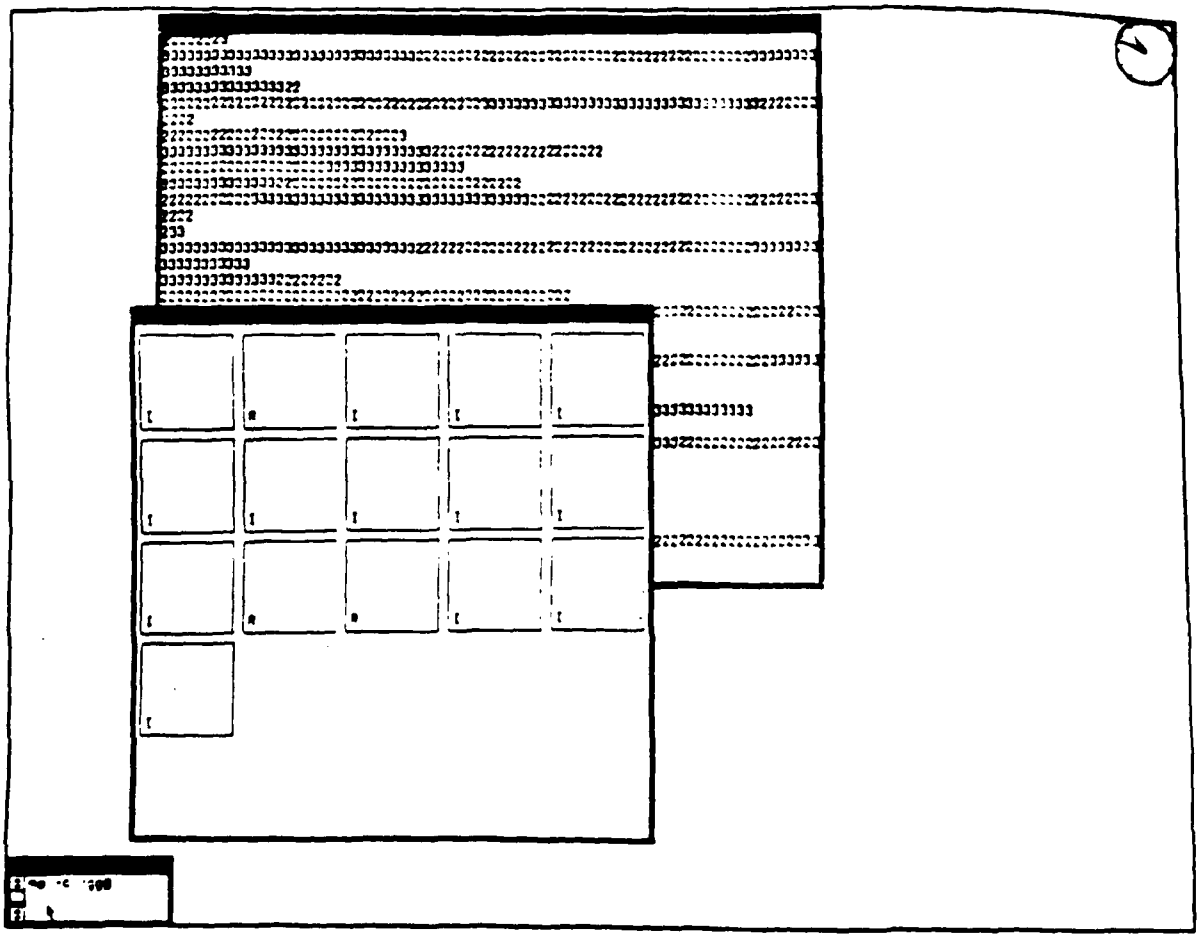

Figure 4. The Multiprocessor Window

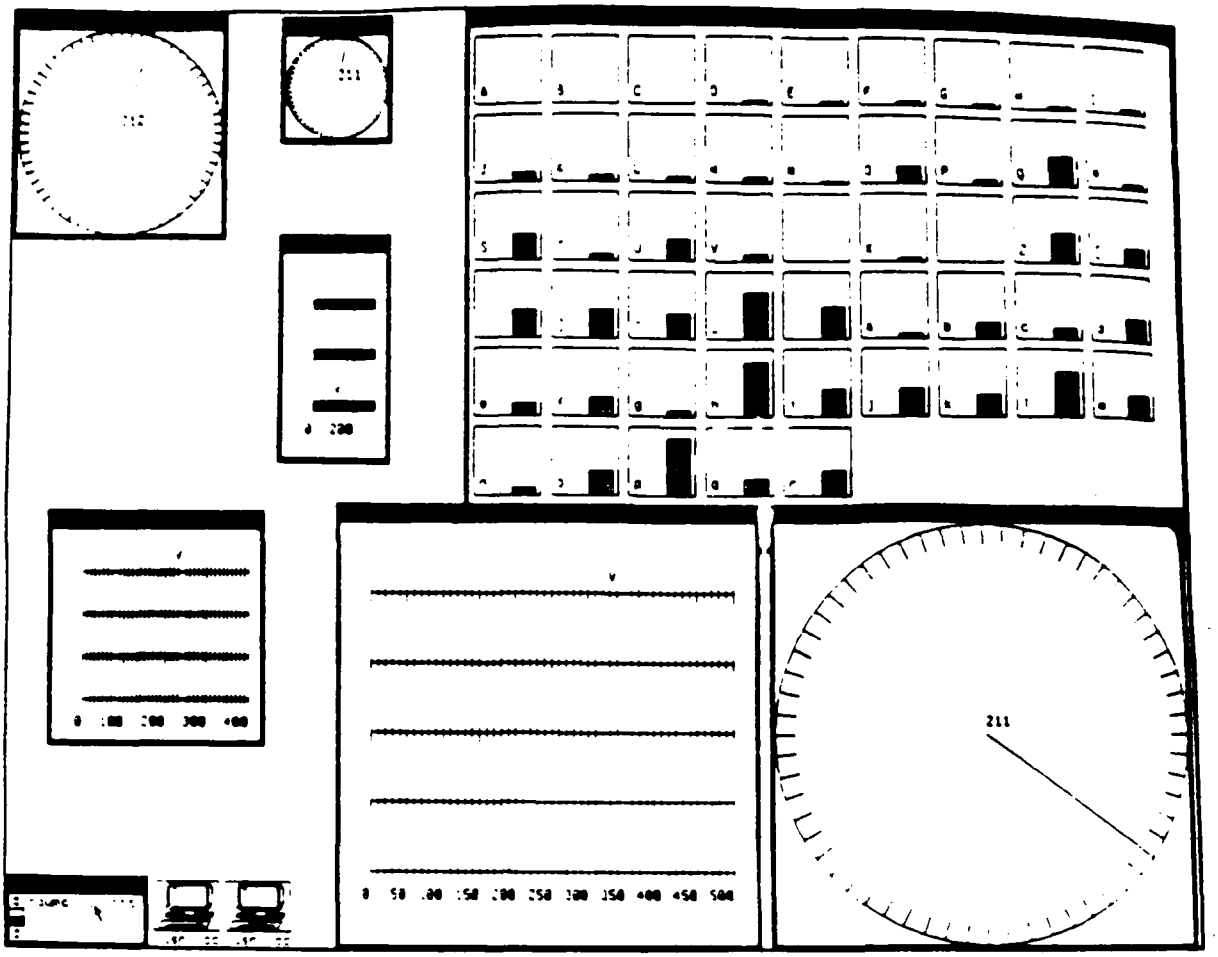

Figure 5. Visualization Abstract Data Types

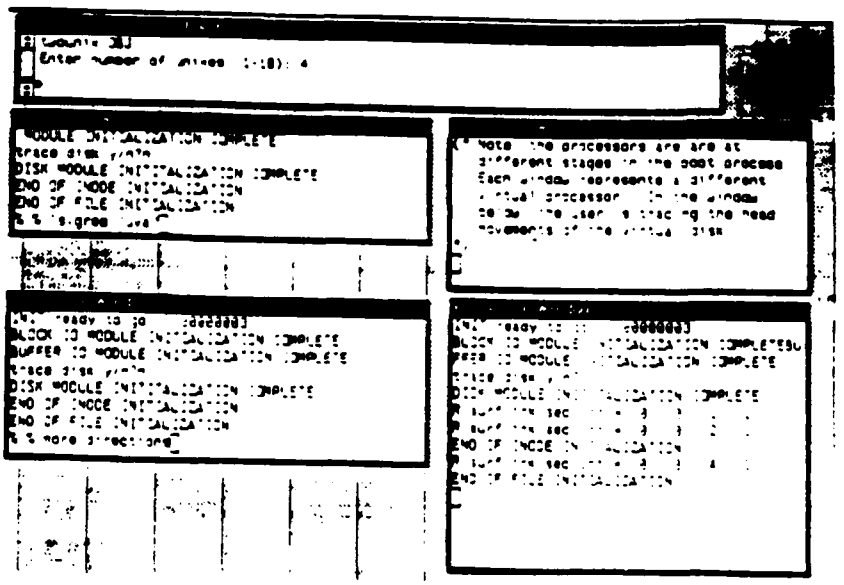

Figure 6. A 4-Node Distributed System

2.4.4 Ethernet Model

Figure 8 illustrates the StarLite windows for EtherNets. As operators, such as "read" or "write," are applied to the simulated devices, the effects are depicted in their windows. For a network device, the network state as well as each packet's size and destination address are displayed. Optionally, the transferred data can also be listed. Each EtherNet window represents a network of as many as several hundred stations. The state of each station is denoted by a letter, with upper case indicating a pending request, and lower case a completed operation. For example, a transmit request remains pending until either the backoff algorithm fails or the transmission completes.

The EtherNet operators are **Transmit**, **Broadcast**, **Receive**, and **receive-All**. There is also an **Idle** state. For example, the EtherNet 2 window shows the current state of a 6-node EtherNet, where the transmission from node five to node two is completed while nodes one, three, four, and six are idle. The standard menu options are **Break, Continue, Speed, Examine, Stop,** and **Exit.** The **Speed** option is particularly useful because it can be used to vary the speed of a physical device so that experiments can be run with different configurations of networks and disks. By increasing the speed of one component we may expose bottlenecks in others.

3.0 Conclusion

StarLite is an effective software education tool. It is the only realistic alternative for students who do not have access to a distributed system. StarLite forces the user to concentrate on the important aspects of a problem by eliminating concern over details. By encouraging the study of elegant and efficient interfaces, the students learn a more disciplined approach to distributed systems development.

References

[1] Liskov, B. et al, Abstraction Mechanisms in CLU. **CACM 20,** 8(Aug 1977), 564-576.

[2] Wegner, P., Programming Languages--Concepts and Research Directions, **Research Directions in Software Technology,** MIT Press, Edited by P. Wegner, (1979) 425-489.

[3] Zimmermann, H., OSI reference Model--The ISO Model of Architecture for Open Systems Interconnection, **IEEE Transactions on Communications COM-28,** (April 1980) 425-432.

[4] Cook, R.P., StarMod, A Language for Distributed Programming, reprinted in **Concurrent programming**, Addison-Wesley, edited by N. Gehani and A.D. McGettrick, (1988).

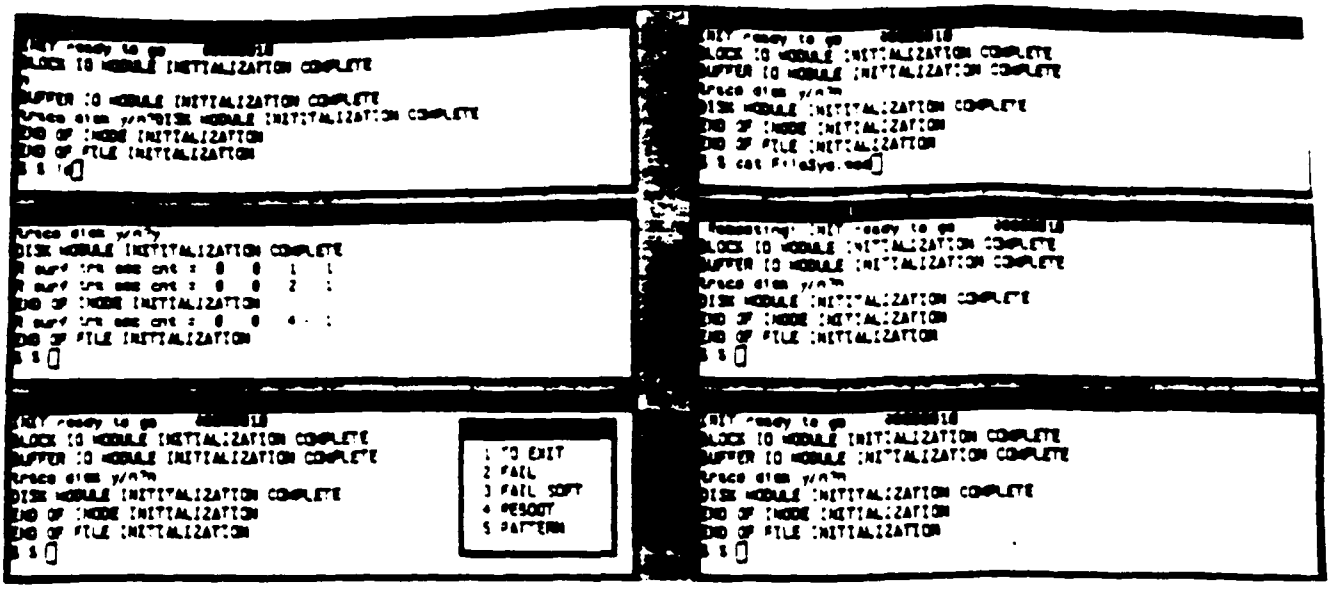

Figure 7. A 6-Node Distributed System

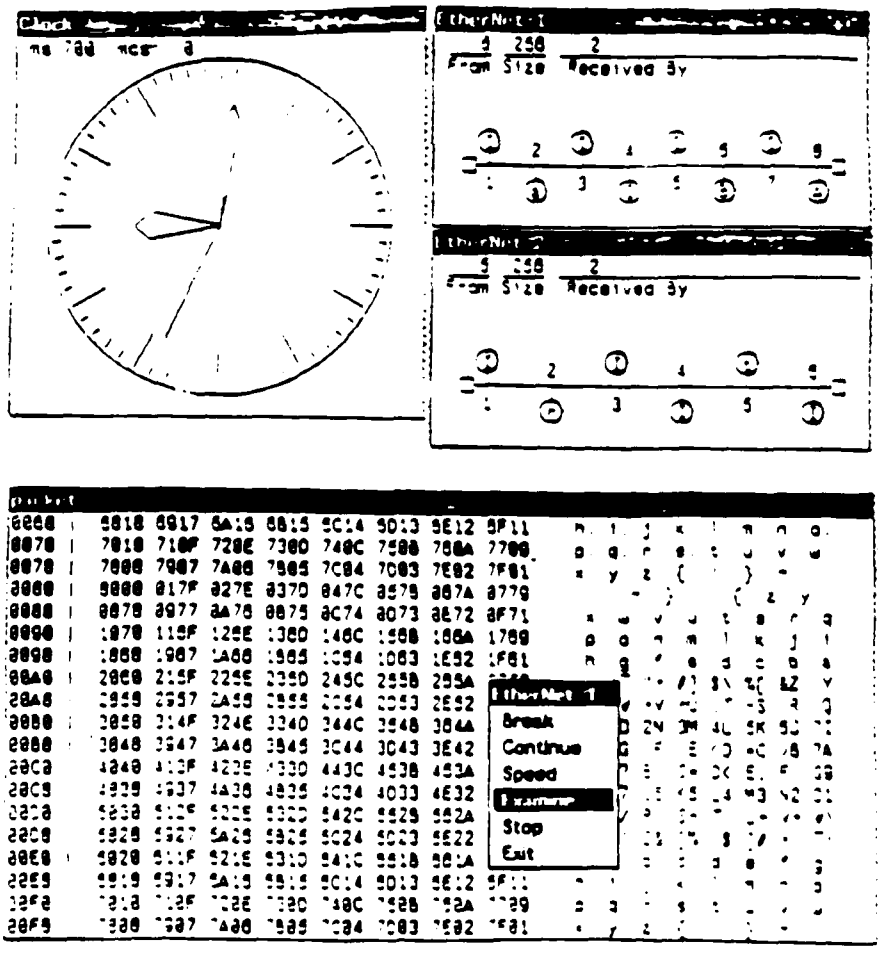

Figure 8. EtherNet Windows

StarLite -- A Laboratory for Operating Systems Research

Abstract

Laboratories are a prerequisite to all scientific investigation. They contain standard equipment that facilitates experimentation. This paper discusses the characteristics of a laboratory for operating systems research. The software executes on a single host yet it supports a variety of machine models, including that of a multi-node distributed system.

1.0 Introduction

In this paper, we report our experiences with a new software laboratory, StarLite. It was constructed to support research in parallel computing, database, operating system, and network technology. The discussion includes the motivation for StarLite, an outline of its structure, and a description of some of its software tools.

Laboratories are a prerequisite to all scientific investigation. They contain standard equipment that facilitates experimentation. The ability to quickly create experiments amplifies a scientist's intellectual ability. The use of visualization tools enhances a scientist's ability to discover the operational characteristics of complex systems.

Another important advantage of standard laboratory equipment is that experiments can be validated independently. In some disciplines, results are not accepted for publication until they have been reproduced by other researchers.

What is an appropriate laboratory for operating systems research? Some common answers are complexity analysis, discrete-event simulation, queuing analysis, IBM's virtual machine environment, remote procedure call packages, and finally physical hardware. We do not claim that StarLite is a replacement for these proven options. Rather, it defines a new dimension that is made possible by advancing software and hardware technology.

StarLite is designed to facilitate systems research. For example, StarLite supports the execution of a distributed or multiprocessor software system on a single hardware processor. In the past, a systems researcher had to purchase multiple CPUs and network interfaces to perform experiments. With StarLite, only a single computer is required. Also, StarLite is portable; the support software is written in C. Portability allows software to be easily shared among different research groups. StarLite provides a standard set of tools and a components library so experiments can be reproduced by other researchers. Finally, software developed using StarLite in a host environment will execute in a given target machine without modification of any layer except the hardware interface. The software is organized as module hierarchies composed from reusable components.

StarLite has been used to implement a multiprocessor UNIX (Phoenix), a distributed database system[1] and a suite of network protocols. It has also been used to support graduate and undergraduate courses in operating systems and database technology.

In the remainder of the paper, the components of StarLite are presented together with a representative sample of software tools. Also, StarLite's effectiveness as a software laboratory

is discussed.

2.0 StarLite Components

The purpose of this Section is to describe the components of the StarLite environment and to explain their use for software development. The StarLite components include a Modula-2 compiler, an interpreter, a window package, a viewer, a simulation package, and a profiler. The compiler and interpreter are implemented in C for portability. The rest of the software is in Modula-2. The system currently runs on SUN workstations and personal computers.

2.1 Compiler

The compiler is compatible with Modula-2 as defined by Wirth[2]. It generates interpreted M-code by default, although it can also generate native code for a range of target machines. As a result, StarLite modules that are developed in the host environment can be retargeted without modification for embedded testing.

2.2 Interpreter

The StarLite interpreter supports the simultaneous execution of multiple virtual processors in a single address space. For example, to test a distributed database system, we might start a file server and several clients. Each node has its own operating system and user-level processes. In the current environment, all the code and data for the virtual machines executes as a single UNIX process. The StarLite interpreter on a SUN 3/260 executes a single virtual processor at a speed of from one to six times that of a PDP 11/40, which was widely used for operating system research ten years ago. Even so, the execution speed is 20-30 times slower than a SUN 3.

To address this speed deficiency, the interpreter also supports mixed-mode execution in which some modules are native code and some interpreted. The native code modules are up to twenty times faster than their equivalent interpreted code versions.

The advantage of the interpreted code versions is that they are machine invariant. As a result, object code developed on any machine can be executed without recompilation on any other. Thus, we have a master components library that is shared, via the network, by all the machines in our building. The invariance is achieved by defining a canonical object module format. The StarLite software tools, such as the Viewer and Profiler, can be used transparently on either class of module.

2.2.1 Machine models

Figure 1 illustrates the three virtual machine models supported by the interpreter: single processor, multiprocessor, and distributed processors. All software developed in the laboratory uses one of the machine models as a base.

The dotted line in the Figure indicates that all of the machine models, including the distributed system, execute as a single UNIX process. The rationale was to make network devices fast by using memory copies for transfers and to make it easy for the software tools to control the state of a distributed computation. The system could easily be extended to use multiple processes or multiple machines, but with a loss of functionality.

For distributed processors, each virtual processor has its own copy of the test software. For the other machine models, the software is shared by all processors.

The "Coroutines" and "Mp" boxes represent the Modula-2 modules that are the test software's view of a machine model. The **definition** modules for Coroutines and Mp define the interfaces

that support interrupt and trap handling, context switches, device I/O, memory management, timer services, and spin locks. These interfaces mimic the C interfaces that would normally be created to make a physical machine's components accessible to an operating system's code.

The Modula-2 **implementation** modules for Coroutines and Mp emulate the behavior of physical machines. For example, a DISABLE procedure call disables all device interrupts.

One goal of StarLite is to retarget to an embedded system only by replacing the implementations of the virtual machine interface modules and recompiling. All higher level software should remain invariant. As a result, the success of StarLite hinges on the design of the virtual machine interfaces (**definition** modules).

If the interfaces are not properly designed, changes will be necessary when retargeting to an embedded system. Changing an interface may require rewriting all dependent modules. If the implementations do not capture the timing or operational aspects of physical devices, empirical analysis may be fruitless.

In addition to supporting the various machine models, the interpreter also implements some other unique features of the StarLite architecture. These include demand loading, clock control, and dynamic restarts.

2.2.2 Demand loading

The StarLite interpreter supports demand loading. That is, modules are loaded at the point that one of their procedures is called. A linker is superfluous. As soon as any module in a test system is compiled, the system can be executed. For example, the operating system (66 modules; 7500 lines) can be compiled and booted in less than 30 seconds.

All test systems begin execution quickly since only the object code for the program module is loaded initially. Additional modules are loaded as they are referenced. For example, one version of the operating system defers loading the file system's modules until a file operation is performed.

2.2.3 Clock control

The existence of race conditions in an embedded system can often make error tracing difficult. For example, inserting an output statement can cause an error to disappear. This effect is possible in both embedded systems and in StarLite. In the former case, it is eliminated by using a hardware monitor or an in-circuit emulation system. In StarLite, we use clock control.

Clock control modifies the interpreter's virtual clocks so that time appears to "stand still." Any number of I/O or debugging actions may occur before resuming execution. This option is also very useful for collecting statistics without disturbing the behavior of a system. Finally, clock control is essential for our visualization aids so that they can be attached to a program without affecting its actions.

2.2.4 Dynamic restart

When debugging an embedded system, it is annoying to discover an error, return to the host level, compile, link, prepare a boot image, reboot, and then run the system to the point of error only to discover another mistake. The problem is magnified for distributed systems.

The StarLite architecture is designed so that an **implementation** module in a running program can be recompiled while the interpreter's execution is suspended. The recompiled module can then be reloaded and the test system restarted without repeating the link-make-boot cycle. This option is used to repair node software while

testing a fault-tolerant distributed system.

Another dynamic restart feature supports the emulation of partial failure as might be experienced in a distributed system. The interpreter allows any loaded module, or set of modules, to be restarted at any time. Thus, reading the corresponding object modules from disk is avoided.

For a distributed system, the user can induce virtual processor failures and then restart operating systems on the failed nodes without loading any software from disk.

2.3 Windows

Distributed systems are characterized by partial failure and by non-deterministic actions. For example, a network device interrupt is non-deterministic because a programmer cannot predict the state of the program when an interrupt occurs. The actions of programs composed from multiple processes may also be non-deterministic.

As a result, visualization aids[3] can be an important component of system development and maintenance support. Multiple text and graphics windows are used in StarLite to highlight and monitor system actions. Any component of a distributed system can be presented to the user as an abstract data type that uses windows to display or modify its actions.

Figure 2 displays instances of the StarLite abstract data types for the clock and disk virtual devices as well as for StateQueues. As device orders, such as "read" or "write" are applied to the virtual devices, the effects are depicted in their corresponding windows.

For a clock, the display shows the elapsed time in terms of the number of clock ticks. For a disk, the surfaces and track position are indicated. The user can also set breakpoints to examine the content of sectors as they are transferred to/from a disk. For a network device, the network state as well as each packet's size, source and destination addresses are displayed. Optionally, the packet content can also be listed.

A StateQueue is a window that displays a single statistic, such as queue length, for a collection of similar entities. The statistic is displayed as a bar chart that is scaled across all the entities. For example, we use a StateQueue to display delay time for all the critical sections in the operating system. The existence of bottlenecks is exposed immediately as a spike.

2.4 Viewer

The StarLite Viewer extends the functionality found in traditional debuggers. First, the Viewer allows the user to explore, monitor and modify any thread, module, procedure, or variable on any processor. Second, the Viewer is an abstract data type. The user can create as many of them as needed by connecting a Viewer to each thread of execution. Also, all hardware details are accessible from a Viewer.

For example, registers or procedure call chains can be examined directly. This is possible because the virtual processor interface is defined as a Modula-2 **definition** module, which in turn is encoded using Gutknecht's symbol file representation[4].

When an instance of the debugger is opened on a coroutine (thread), it displays Message, Module, Coroutine, and Control Panel windows. The Module window is a scrollable list of the modules that comprise a thread's implementation. The source code for a module, such as TreeDemo, is displayed if it is selected using the mouse. The Source window is also scrollable. The user sets breakpoints by clicking on text. Breakpoints, such as the one on line 58, are then displayed in the Control Panel window.

By clicking on the **continue** option, execution is resumed until the next breakpoint is met. At that point, the Coroutine window is updated to display the current state vector and a scrollable, call-chain list.

If the breakpoint is at the module level, as in the example, its global variables are displayed symbolically. If in a procedure, local variables are displayed. By clicking on data names, the user can view and modify program variables, either at the module global level or anywhere in the call chain. If a coroutine variable is examined, the Viewer switches to a new call chain automatically.

Two other features of the Viewer are its openess and its support of **type** filters. The StarLite Viewer is "open" because all its capabilities are available under program control. For example, a running program can set breakpoints on itself by using the procedural interface to the debugger..

A **type** filter is a module that controls Viewer access to **type** instances. As a result, users can tailor visualizations of data to create their own debugging "views". Figure 3 illustrates the use of a type filter to examine a tree manipulation program. The filter displays the tree data structure graphically rather than as a collection of fields.

Another unique feature of StarLite filters is that they can retain the interactive features of the underlying Viewer. In the Figure, two of the tree nodes have been selected with the mouse. One is the leaf node 'g' and the other is the interior node 'i'. The filter causes the debugger to display their field data interactively. Next, the right subtree of node 'i' is selected. Since this is also a Tree, it too is displayed using the filter.

2.5 Simulation package

In most cases, the correctness of a test system does not depend on time. For instance, file systems do not usually have timing constraints. However for empirical analysis, it is useful to manipulate virtual devices that have timing characteristics that emulate those of the corresponding physical devices. The StarLite simulation package provides an interface that captures much of the functionality of a discrete-event simulation language, such as GPSS.

The entities provided are a simulation time clock, tables, and stores. The library also contains a variety of statistics and plotting routines to support the analysis of the resulting data.

2.6 Profiler

Figure 4 shows the profiling tool in use with the Phoenix operating system. The profiler can be used with any of the machine models to analyze the performance of sequential, parallel, or distributed algorithms.

The top window, Modules, lists a frequency distribution by module number of where an executing system, in this case the Phoenix operating system, is spending its time. As spikes appear, the user can click on a module in the Module Key window to display additional histograms. In the Figure, the BitMap and PROCESS modules have been selected. The module distributions are by program counter value.

We are working on an enhanced Profiler that will display source text when a program counter value is selected. At present, the inversion must be done by a separate program.

The bottom window, Instructions, records a frequency distribution by opcode value of the instructions being executed by the virtual machine architecture. Each histogram adjusts its scale dynamically.

3.0 Two Examples

In this Section, two examples are used to demonstrate some of the uses of StarLite. The first

example uses a distributed system, the second a a multiprocessor.

The first example, in Figure 5, is a fault-tolerant implementation of a one-bit sliding window protocol running on a 4-node distributed system. Processors 1 and 4 are linked as well as 2 and 3 to create two connections.

Each virtual processor has its own memory, interrupt and trap vectors, clock, and a connection to a shared EtherNet device. As each copy of the protocol executes, it lists the number of packets received, the control bits, and the packet sequence number. In the absence of errors, one packet is received for every packet that is passed up to the user. Only "in sequence" packets are listed; thus, a receive count greater than one indicates a mismatched timer interval or the one-bit protocol synchronization problem[5] because more packets than expected are being received.

The Control Panel allows the user to insert processor failures or restart actions into a running system. FAIL stops the selected processor immediately. FAILSOFT generates a power fail interrupt and, after a fixed period, stops the processor. FAILSOFT is used to test algorithms that depend on stable storage. REBOOT will reboot a selected program on a failed node. In Figure 5, processor 3 has been rebooted once, which is denoted by its "3.1" label. Notice that the packet sequence numbers in processors 2 and 3 are different. When processor 3 rebooted, it recovered the sequence number stream so that processor 2 did not need to restart the packet sequence. Finally, PATTERN automatically fails and reboots nodes based on a user-specified failure function.

The second example in Figure 6 illustrates the Phoenix operating system running on a 16-node multiprocessor. We are using pipes at the shell level to test our cache-affinity scheduling algorithm. A cache affinity processor assignment algorithm attempts to schedule a process on the processor most likely to have the largest amount of retained data/instructions in its cache.

In the example, two processes communicating through a pipe block on the "pipe full" and "pipe empty" conditions. Normally, when a process blocks, it can be restarted on any processor. However, if each processor has data, instruction, or address caches, restarting on a different processor can incur a significant performance penalty.

The display tool in Figure 6 depicts which processors are idle or busy as the system executes. As a result, we get a graphic interpretation of how well an affinity algorithm performs. Lots of movement is bad; a steady view is good.

4.0 Summary

We cannot offer proof that the StarLite laboratory is "the" appropriate laboratory for operating systems research. However, it does demonstrate that it is feasible to pursue major systems projects in a virtual interface environment. The environment cannot execute programs as fast as a physical machine and it would be infeasible to emulate all of a physical machine's effects, such as memory interference. However, the advantages are a greatly accelerated development cycle and totally portable, and hence reproducible, results. Furthermore, developed software can be used immediately in classes.

StarLite is feasible because workstations now have large physical memories and are fast enough to run interpreters at the speed of physical machines ten years ago. Ten years ago it would not have been feasible to run an emulator on a PDP-11 and then to implement an operating system on top of it. Today it is feasible. StarLite does not currently take advantage of the multithread support available on some of the newer workstations, but it could.

StarLite demonstrates the effectiveness of research based on virtual interfaces as opposed to IBM's virtual machine approach, or even the

traditional use of physical machines. The great majority of the machine details contribute nothing but a source of error to the intellectual effort of improving operating system technology. StarLite forces the user to concentrate on the import aspects of a problem by eliminating detail.

By encouraging the study of elegant and functionally complete interfaces, we impose discipline on hardware designers who all too often make all the decisions before the operating system designer sees the machine. In the future, the operating system designer will dictate the constraints and will be able to defend his actions since StarLite supports operating system development for virtual hardware.

References

[1] Son, S.H. and R.P. Cook, Scheduling and Consistency in Real-Time Database Systems, **Sixth IEEE Workshop on Real-Time Software and Operating Systems**, (May 1989) 42-45.

[2] Wirth, N., **Programming in Modula-2**, Springer-Verlag, (1982, 1983).

[3] Fowler, R., T. LeBlanc, J. Mellor-Crummey, An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors, **SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging**, (May 1988) 163-173.

[4] Gutknecht, J., Separate Compilation in Modula-2: An Approach to Efficient Symbol Files, **IEEE Software 3**, 6(Nov. 1986) 29-38.

[5] Tanenbaum, A.S., **Computer Networks**, Prentice-Hall, (1981).

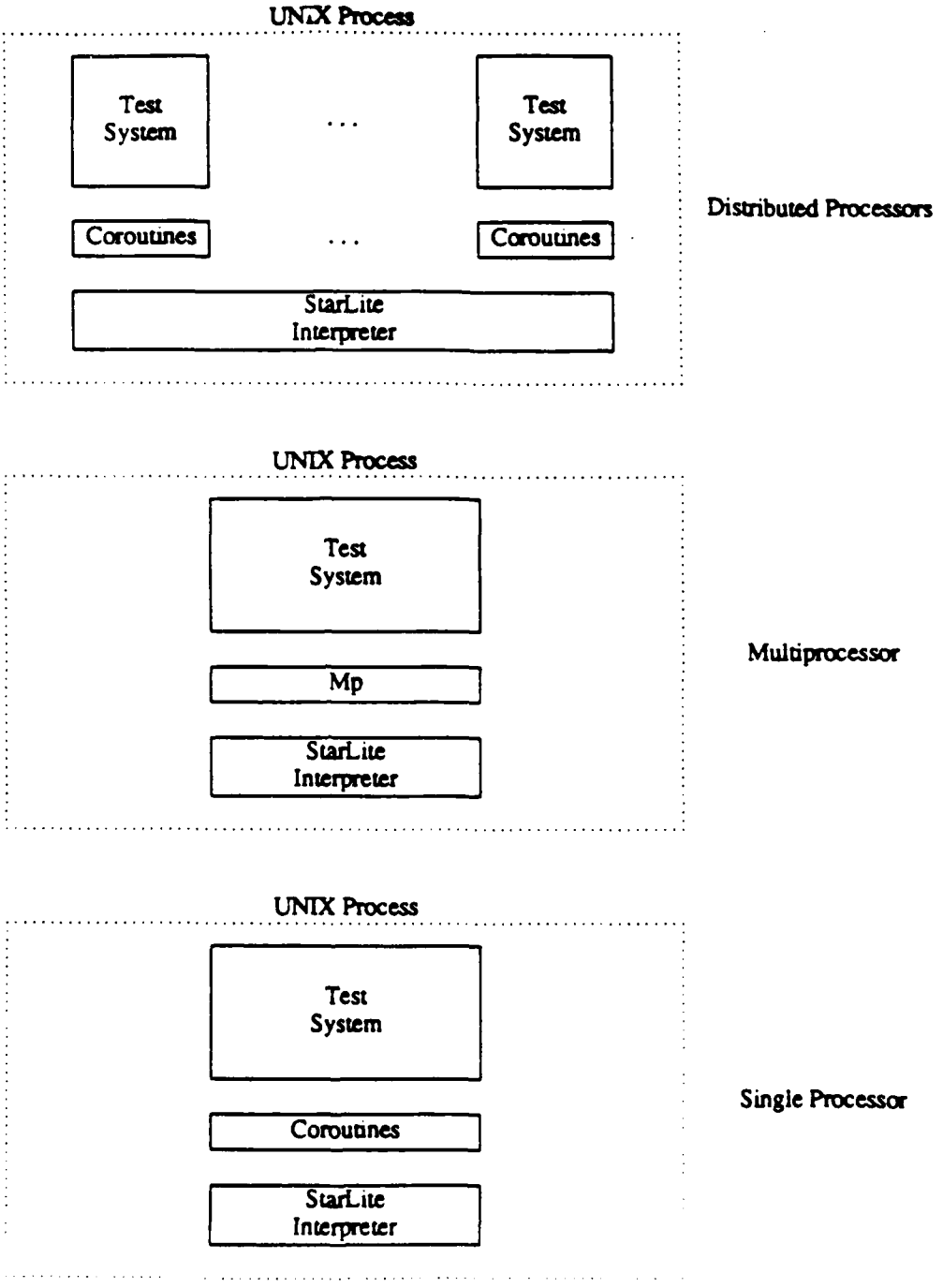

Figure 1. Machine Models Supported by StarLite

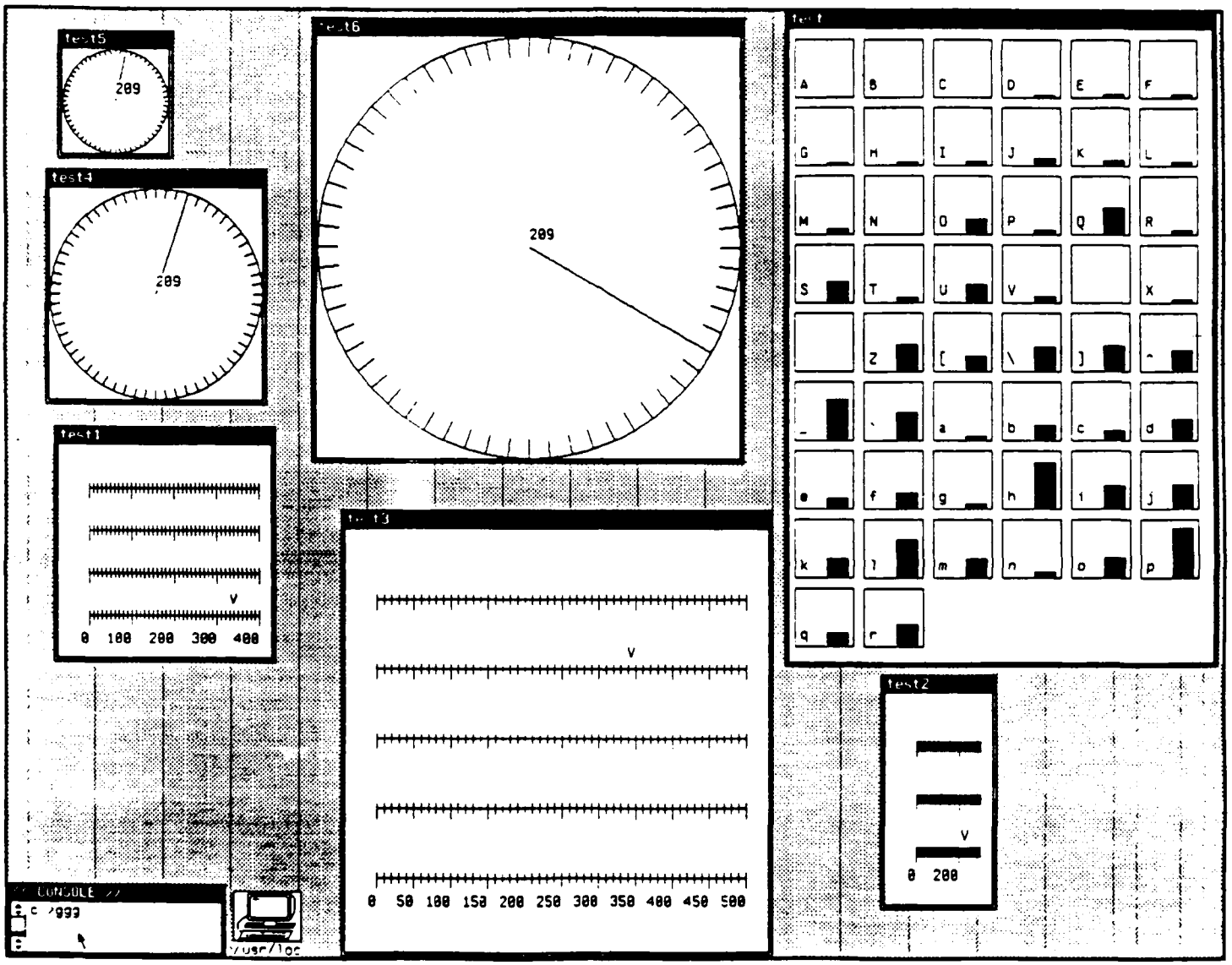

Figure 2. Virtual Devices and State Queues

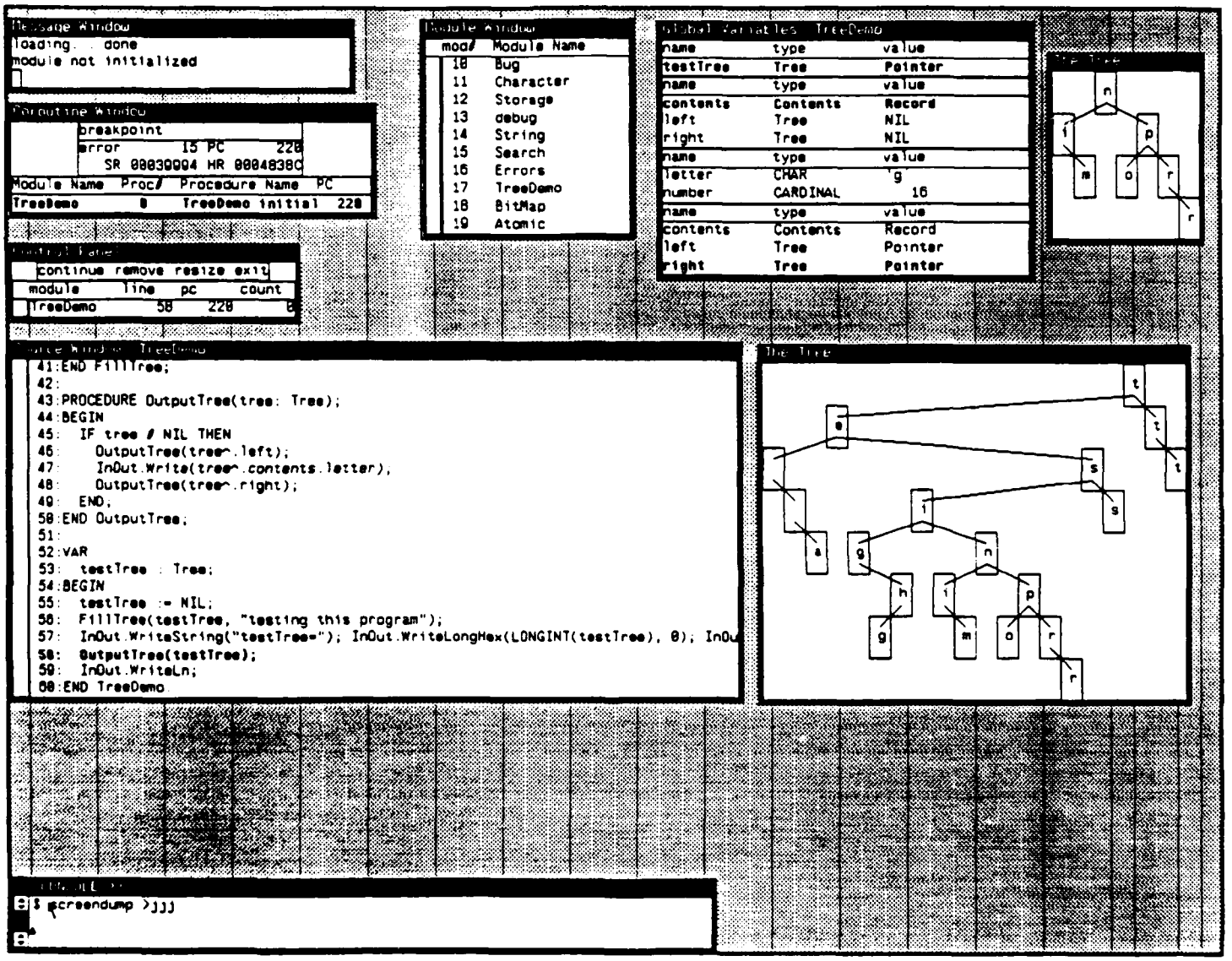

Figure 3. The StarLite Viewer

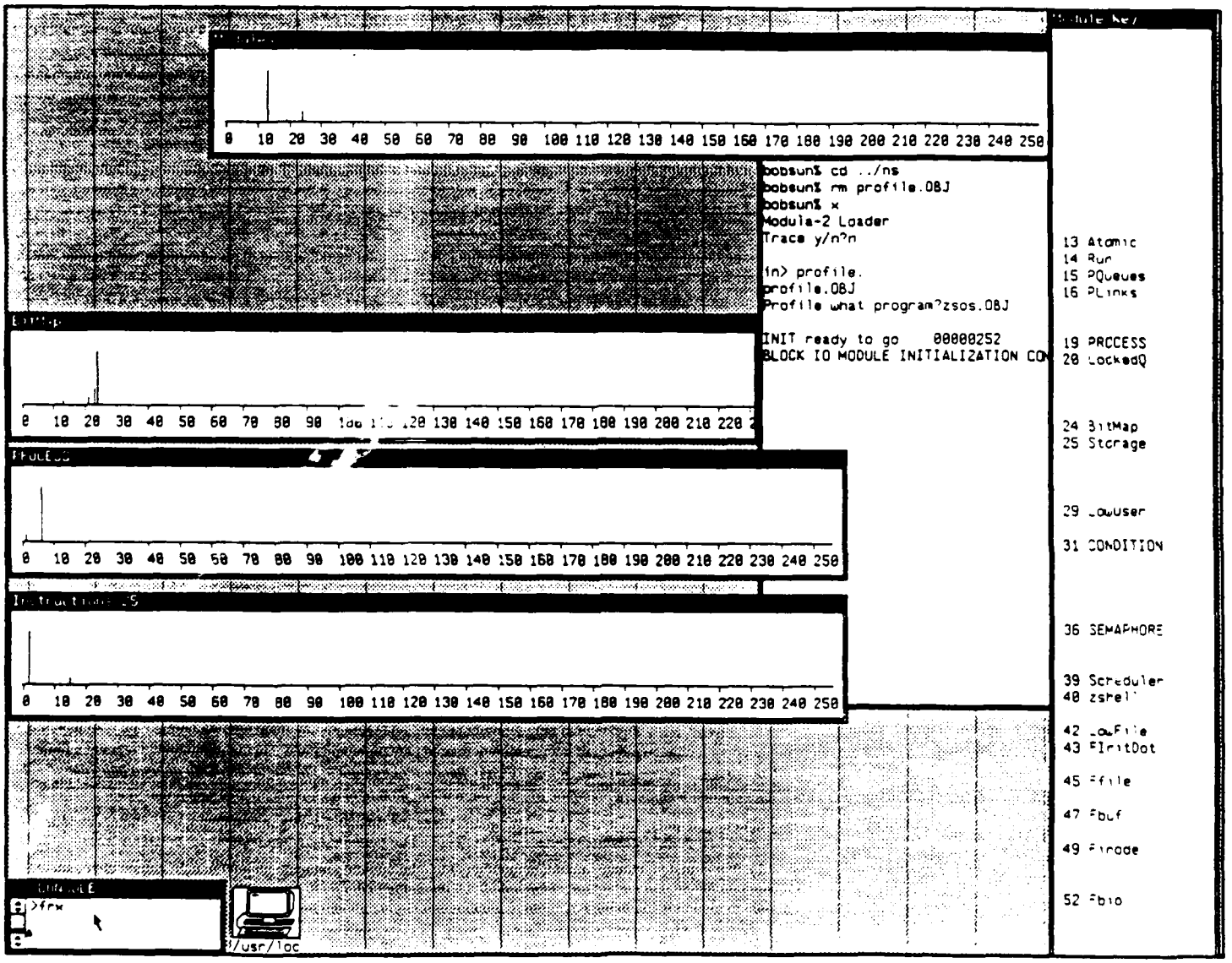

Figure 4. The StarLite Profiler

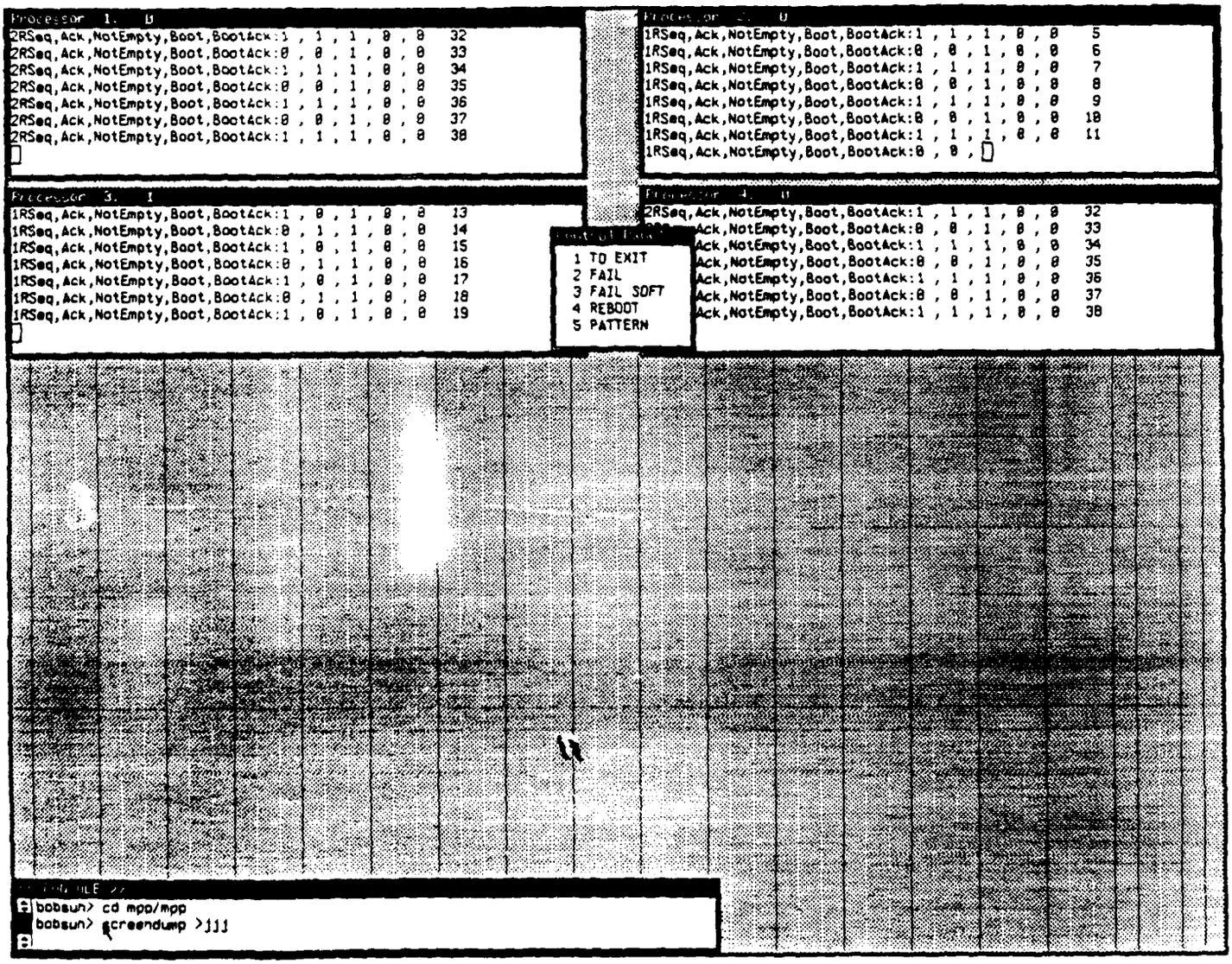

Figure 5. One-Bit Sliding Window Protocol, 4 Nodes, 2 Connections

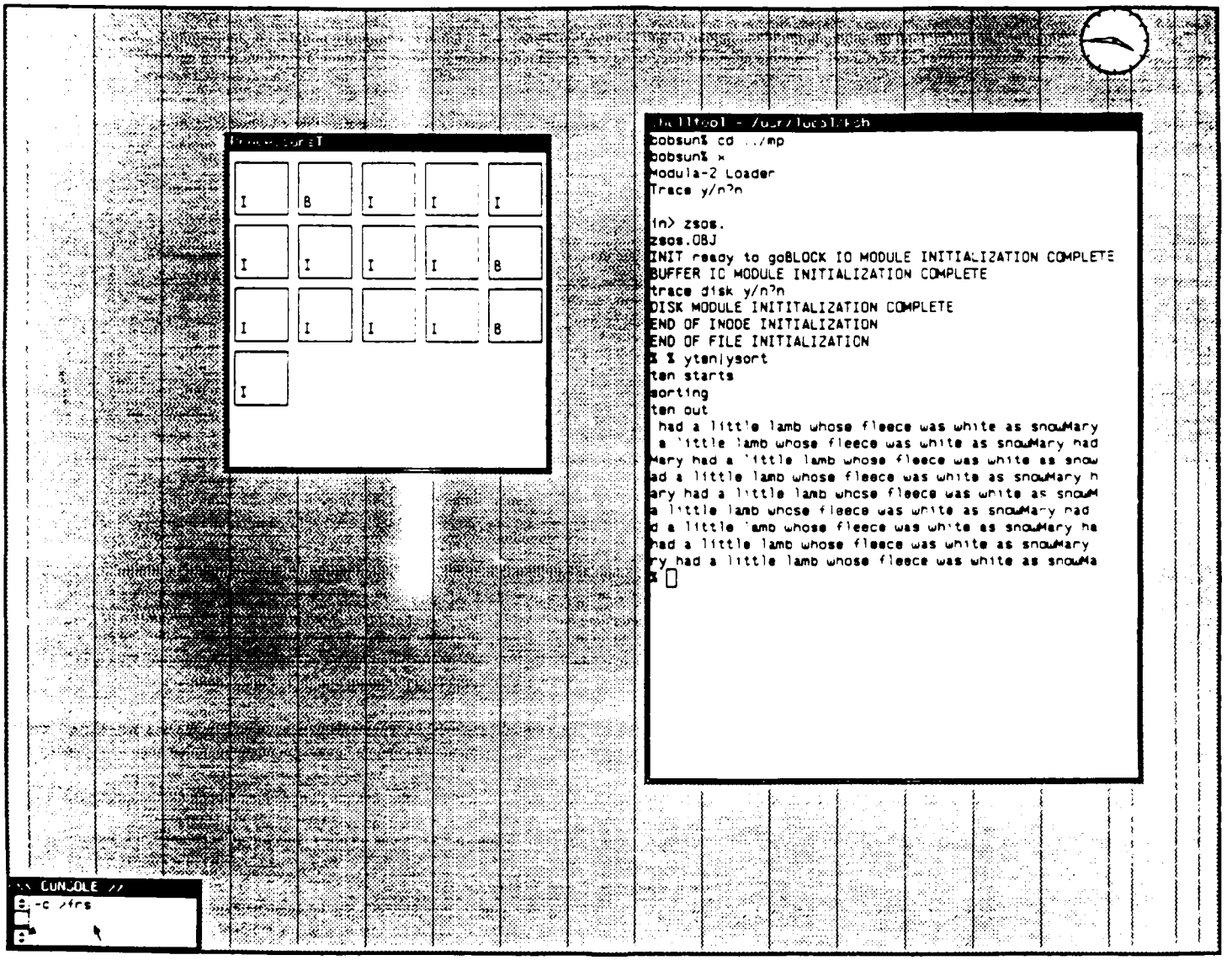

Figure 6. Phoenix Running on a 16-Node Multiprocessor

The StarLite Programming Environment

Extended Abstract

1.0 Introduction

In this paper, we discuss the components and design of the StarLite programming environment for Modula-2[1]. StarLite was designed to facilitate systems research in parallel computing, database, operating system, and network technology.

For example, StarLite supports the execution of a distributed or multiprocessor software system on a single hardware processor. In the past, a systems researcher had to purchase multiple CPUs and network interfaces to perform experiments. With StarLite, only a single computer is required. Also, StarLite is portable; the support software is written in C. Portability allows software to be easily shared among different research groups. StarLite provides a standard set of tools and a components library so experiments can be reproduced by other researchers. Finally, software developed using StarLite in a host environment will execute (after retargeting) in a given embedded environment without modification of any layer except the hardware interface. The software is organized as module hierarchies composed from reusable components.

StarLite has been used to implement a multiprocessor UNIX (Phoenix), a distributed database system[2] and a suite of network protocols. It has also been used to support graduate and undergraduate courses in operating systems and database technology. In the remainder of the paper, the components of StarLite are presented together with a brief introduction to the challenges encountered in their design.

2.0 StarLite Components

The StarLite components include a Modula-2 compiler, an interpreter, a window package, a viewer, a simulation package, a profiler, and a software reuse library. The compiler and interpreter are implemented in C for portability. The rest of the software is in Modula-2. The system currently runs on SUN 3/4 workstations and personal computers.

2.1 Compiler

The compiler is compatible with Modula-2 as defined by Wirth. It generates interpreted M-code by default, although it can also generate native code for a range of target machines. As a result, StarLite modules that are developed in the host envi-

ronment can be retargeted without modification for embedded testing. The compilation time on a SUN 3/260 for the 66 modules (7500) lines that comprise the Phoenix operating system is one minute (clock) or 16 seconds (user) time.

One of the goals for StarLite is object module, execution invariance across all architectures. Thus, the object code in our shared library can be executed on any architecture without recompilation. Execution invariance is achieved by defining a canonical object module format, designing special instructions, and by using a "smart" loader. For example, the loader converts every value in the constant segment of an object module to match the byte-ordering and floating-point format of the target.

2.2 Interpreter

The StarLite interpreter supports the simultaneous execution of multiple virtual processors in a single address space. For example, to test a distributed database system, we might start a file server and several clients. Each node has its own operating system and user-level processes. In the current environment, all the code and data for the virtual machines executes as a single UNIX process. The StarLite interpreter on a SUN 3/260 executes a single virtual processor at a speed of from one to six times that of a PDP 11/40, which was widely used for operating system research ten years ago. Even so, the execution speed is 20-30 times slower than a SUN 3.

To address this speed deficiency, the interpreter also supports mixed-mode execution in which some modules are native code and some interpreted. The native code modules are up to twenty times faster than their equivalent interpreted code versions. The transition from interpreted code to native code is accomplished by means of an XM (exchange machine) instruction in the interpreter. The native code is generated to conform to the StarLite virtual machine architecture so that software tools, such as the Viewer and Profiler, can be used transparently on either class of module.

2.2.1 Machine models

Figure 1 illustrates the three virtual machine models supported by the interpreter: single processor, multiprocessor, and distributed processors. All software uses one of the machine models as a base.

The dotted line in the Figure indicates that all of the machine models, including the distributed system, execute as a single UNIX process. The rationale was to make network devices fast by using memory copies for transfers and to make it easy for the software tools to control the state of a distributed computation. The system could easily be extended to use multiple processes or multiple machines, but with a loss of functionality. For distributed processors, each virtual processor has its own copy of the test

software. For the other machine models, the software is shared by all processors. Figure 2 illustrates the Phoenix operating system running on a 16-node multiprocessor together with a visualization tool that indicates whether a processor is busy or idle.

The "Coroutines" and "Mp" boxes represent the Modula-2 modules that are the test software's view of a machine model. The **definition** modules for Coroutines and Mp define the interfaces that support interrupt and trap handling, context switches, device I/O, memory management, timer services, and spin locks. These interfaces mimic the C interfaces that would normally be created to make a physical machine's components accessible to an operating system's code. The Modula-2 **implementation** modules for Coroutines and Mp emulate the behavior of physical machines. For example, a DISABLE procedure call disables all device interrupts.

One goal of StarLite is to retarget to an embedded system only by replacing the implementations of the virtual machine interface modules and recompiling. All higher level software should remain invariant. As a result, the success of StarLite hinges on the design of the virtual machine interfaces (**definition** modules). If the interfaces are not properly designed, changes will be necessary when retargeting to an embedded system. Changing an interface may require rewriting all dependent modules. If the implementations do not capture the timing or operational aspects of physical devices, empirical analysis may be fruitless.

In addition to supporting the various machine models, the interpreter also implements some other unique features of the StarLite architecture. These include demand loading, clock control, and dynamic restarts.

2.2.2 Demand loading

The StarLite interpreter supports demand loading. That is, modules are loaded at the point that one of their procedures is called or when an external variable is referenced. A linker is superfluous; however, "cat" can be used for that purpose if desired. As soon as any module in a test system is compiled, the system can be executed. All test systems begin execution quickly since only the object code for the program module is loaded initially. Additional modules are loaded as they are referenced. Implementing demand loading is complicated by Modula-2's module initialization conventions.

2.2.3 Clock control

The existence of race conditions in an embedded system can often make error tracing difficult. For example, inserting an output statement can cause an error to disappear. This effect is possible in both embedded systems and in StarLite. In the former case,

it is eliminated by using a hardware monitor or an in-circuit emulation system. In StarLite, we use clock control.

Clock control modifies the interpreter's virtual clocks so that time appears to "stand still." Any number of I/O or debugging actions may occur before resuming execution. This option is also very useful for collecting statistics without disturbing the behavior of a system. Finally, clock control is essential for our visualization aids so that they can be attached to a program without affecting its actions.

The StarLite clock is driven by instruction execution. At present, one tick represents 100 instructions. For algorithm or system analysis, we get an absolute measure (in number of instructions) of every improvement.

2.2.4 Dynamic restart

When debugging an embedded system, it is annoying to discover an error, return to the host level, compile, link, prepare a boot image, reboot, and then run the system to the point of error only to discover another mistake. The problem is magnified for distributed systems. The StarLite architecture is designed so that an **implementation** module in a running program can be recompiled while the interpreter's execution is suspended. The recompiled module can then be reloaded and the test system restarted without repeating the link-make-boot cycle. This option is used to repair node software while testing a fault-tolerant distributed system.

Another dynamic restart feature supports the emulation of partial failure as might be experienced in a distributed system. The interpreter allows any loaded module, or set of modules, to be restarted at any time. Thus, reading the corresponding object modules from disk is avoided. For a distributed system, the user can induce virtual processor failures and then restart operating systems on the failed nodes without loading any software from disk.

2.3 Viewer

The StarLite Viewer extends the functionality found in traditional debuggers. First, the Viewer allows the user to explore, monitor and modify any thread, module, procedure, or variable on any processor. Second, the Viewer is an abstract data type. The user can create as many of them as needed by connecting a Viewer to each thread of execution. Also, all hardware details are accessible from a Viewer. For example, registers or procedure call chains can be examined directly. This is possible because the virtual processor interface is defined as a Modula-2 **definition** module, which in

turn is encoded using Gutknecht's symbol file representation[3].

When an instance of the debugger is opened on a coroutine (thread), it displays Message, Module, Coroutine, and Control Panel windows. The Module window is a scrollable list of the modules that comprise a thread's implementation. The source code for a module, such as TreeDemo, is displayed if it is selected using the mouse. The Source window is also scrollable. The user sets breakpoints by clicking on text. Breakpoints, such as the one on line 58, are then displayed in the Control Panel window. By clicking on the **continue** option, execution is resumed until the next breakpoint is met. At that point, the Coroutine window is updated to display the current state vector and a scrollable, call-chain list.

If the breakpoint is at the module level, as in the example, its global variables are displayed symbolically. If in a procedure, local variables are displayed. By clicking on data names, the user can view and modify program variables, either at the module global level or anywhere in the call chain. If a coroutine variable is examined, the Viewer switches to a new call chain automatically.

Two other features of the Viewer are its openess and its support of **type** filters. The StarLite Viewer is "open" because all its capabilities are available under program control. For example, a running program can set breakpoints on itself by using the procedural interface to the debugger..

A **type** filter is a module that controls Viewer access to **type** instances. As a result, users can tailor visualizations of data to create their own debugging "views". Figure 3 illustrates the use of a type filter to examine a tree manipulation program. The filter displays the tree data structure graphically rather than as a collection of fields.

Another unique feature of StarLite filters is that they can retain the interactive features of the underlying Viewer. In the Figure, two of the tree nodes have been selected with the mouse. One is the leaf node 'g' and the other is the interior node 'i'. The filter causes the debugger to display their field data interactively. Next, the right subtree of node 'i' is selected. Since this is also a Tree, it too is displayed using the filter.

2.4 Profiler

Figure 4 shows the profiling tool in use with the Phoenix operating system. The profiler can be used with any of the machine models to analyze the performance of sequential, parallel, or distributed algorithms.

The top window, Modules, lists a frequency distribution by module number of where an executing system, in this case the Phoenix operating system, is spending its time.

As spikes appear, the user can click on a module in the Module Key window to display additional histograms. In the Figure, the BitMap and PROCESS modules have been selected. The module distributions are by program counter value.

We are working on an enhanced Profiler that will display source text when a program counter value is selected. At present, the inversion must be done by a separate program. The bottom window, Instructions, records a frequency distribution by opcode value of the instructions being executed by the virtual machine architecture. Each histogram adjusts its scale dynamically.

2.5 The software reuse library

The StarLite Reuse Library currently contains several hundred modules (100,000 lines) that are all implemented as abstract data types. Modula-2 is our vehicle to capture existing software technology. All students, both graduate and undergraduate, and faculty who use the system contribute to the library. A module must pass rigorous standards to be accepted for shared use.

In addition to the traditional, Stack and Queue ADTs, the library provides modules for multiple machine, language, and computational models. For example, we have a Prolog inference engine and a relational database package that can be called from Modula-2 programs. We also have modules for concurrent programming, exception handling, locking, rendezvous and promise-based message passing, and discrete-event simulation.

3.0 Summary

StarLite is feasible because workstations now have large physical memories and are fast enough to run interpreters at the speed of physical machines ten years ago. Ten years ago it would not have been feasible to run an emulator on a PDP-11 and then to implement an operating system on top of it. Today it is feasible. StarLite does not currently take advantage of the multi-thread support available on some of the newer workstations, but it could.

References

[1] Wirth, N., **Programming in Modula-2**, Springer-Verlag, (1982, 1983).

[2] Son, S.H. and R.P. Cook, Scheduling and Consistency in Real-Time Database Systems, **Sixth IEEE Workshop on Real-Time Software and Operating Systems**, (May 1989) 42-45.

[3] Gutknecht, J., Separate Compilation in Modula-2: An Approach to Efficient Symbol Files, **IEEE Software 3**, 6(Nov. 1986) 29-38.

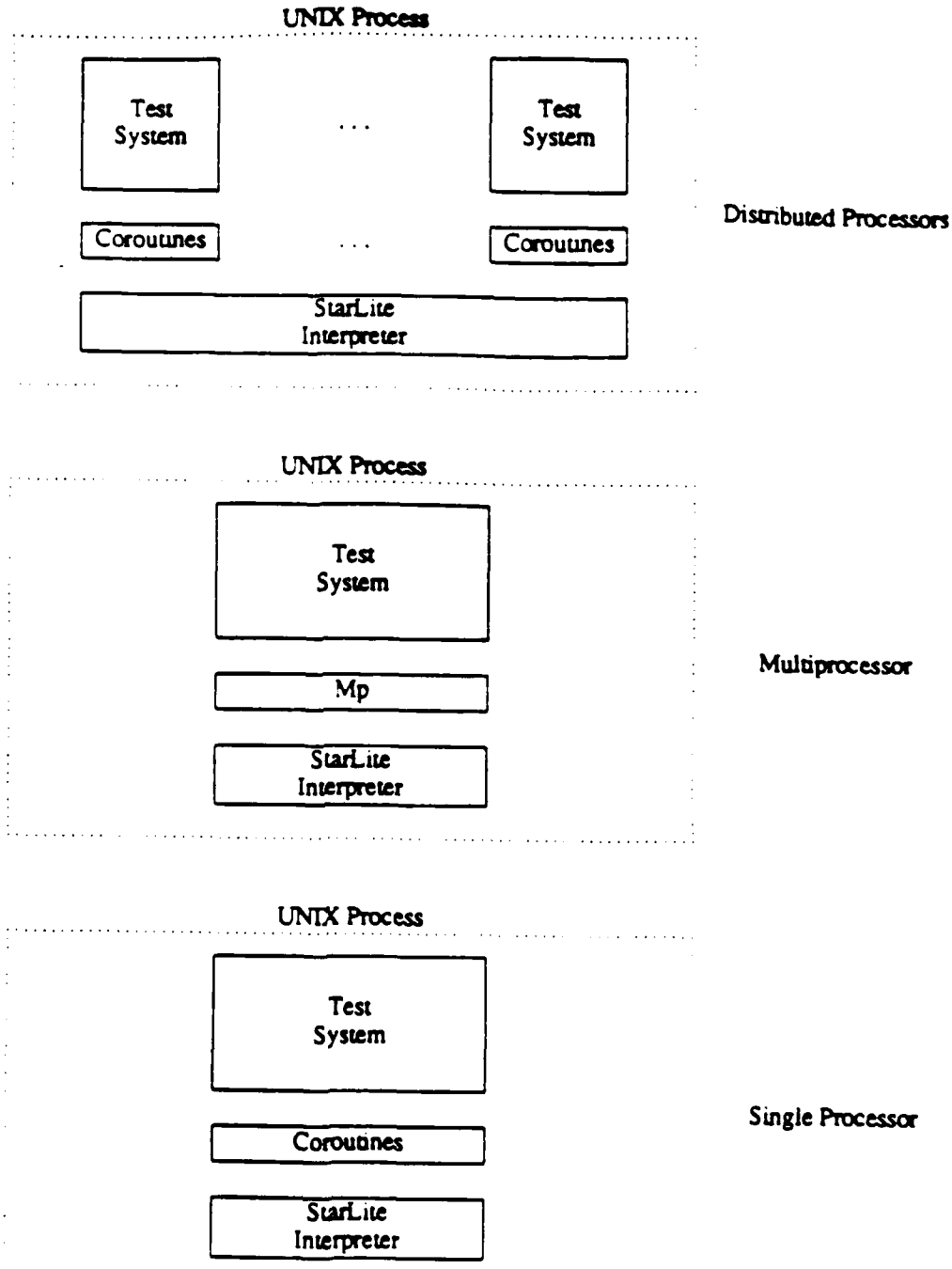

Figure 1. Machine Models Supported by StarLite

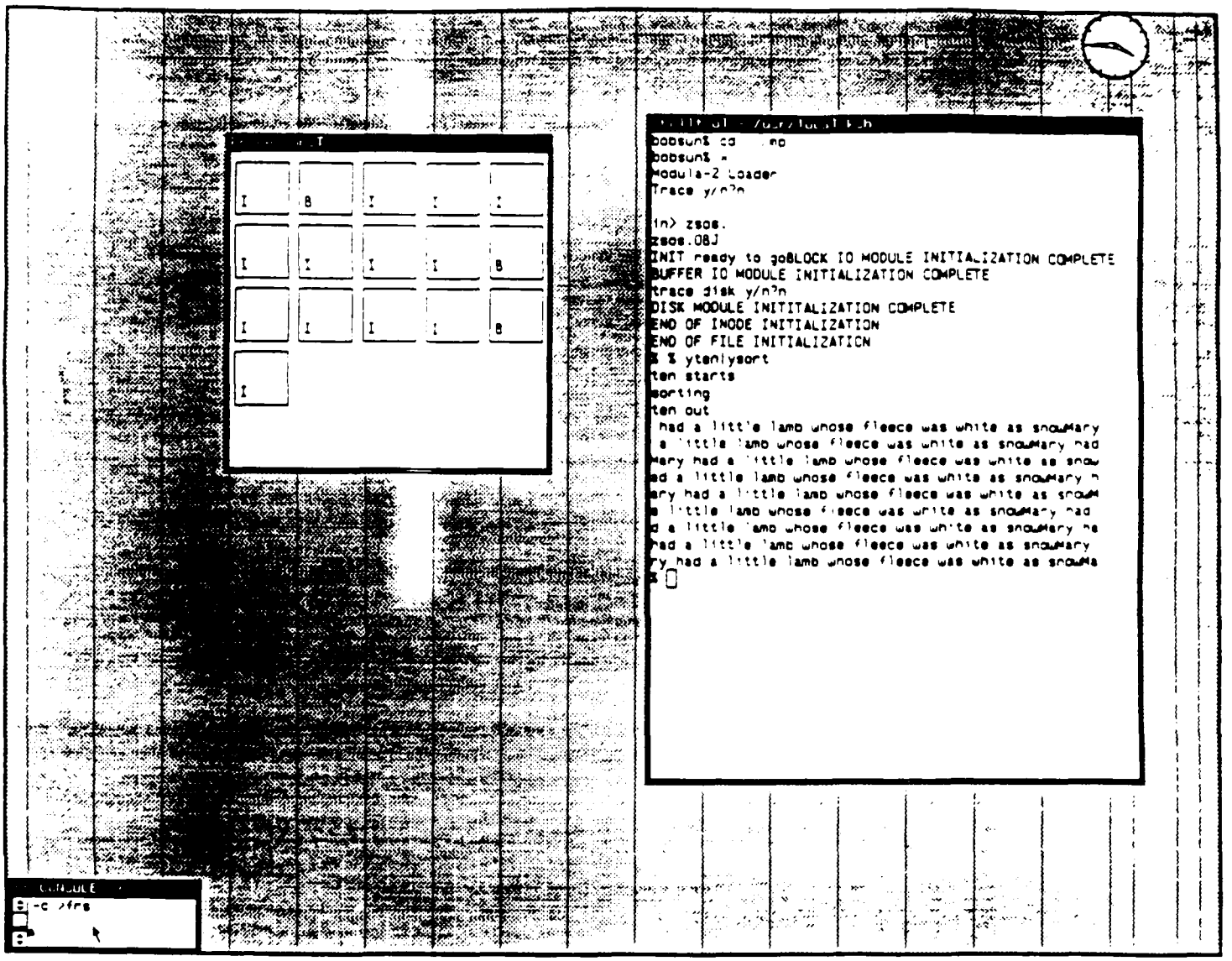

Figure 2. Phoenix Running on a 16-Node Multiprocessor

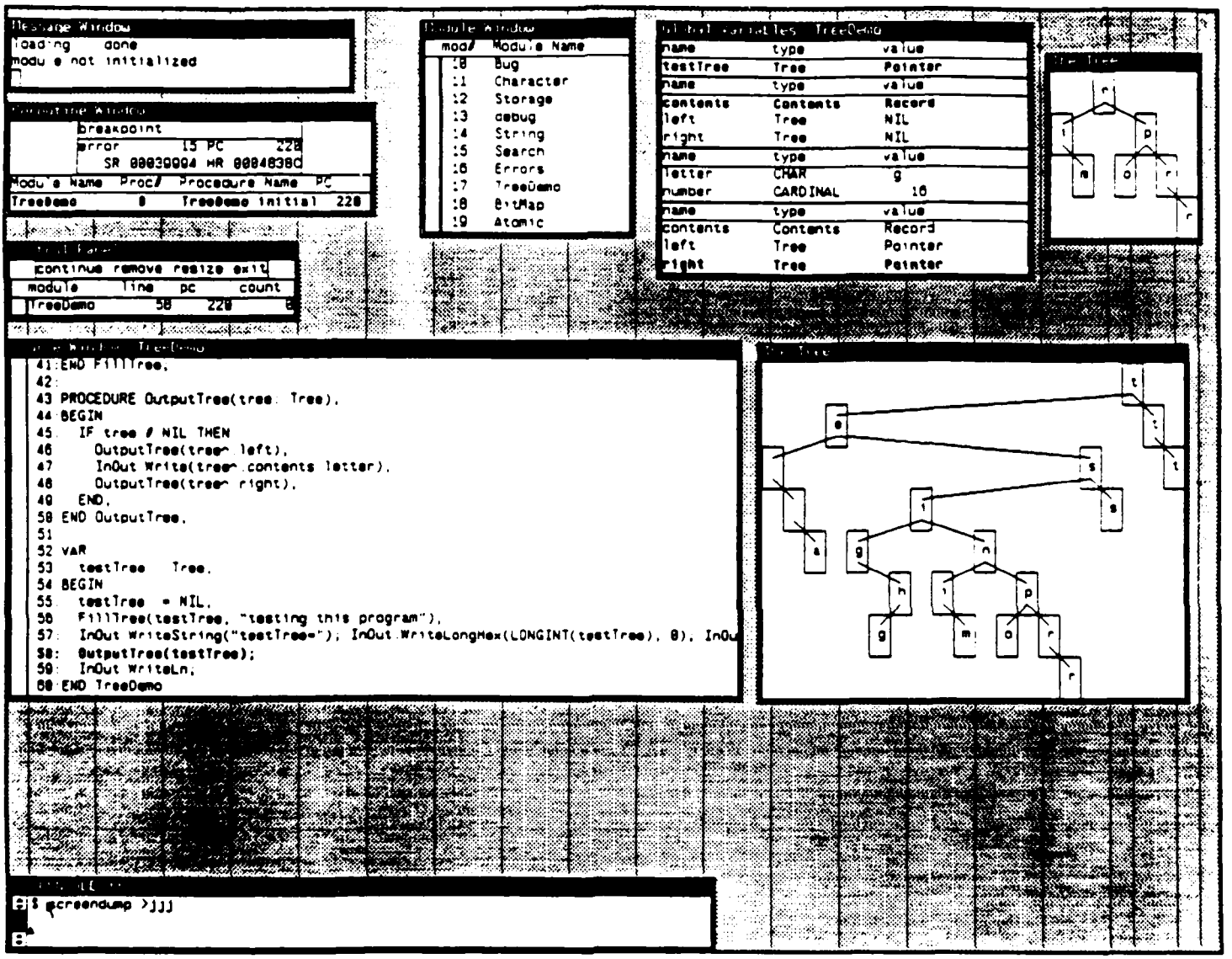

Figure 3. The StarLite Viewer

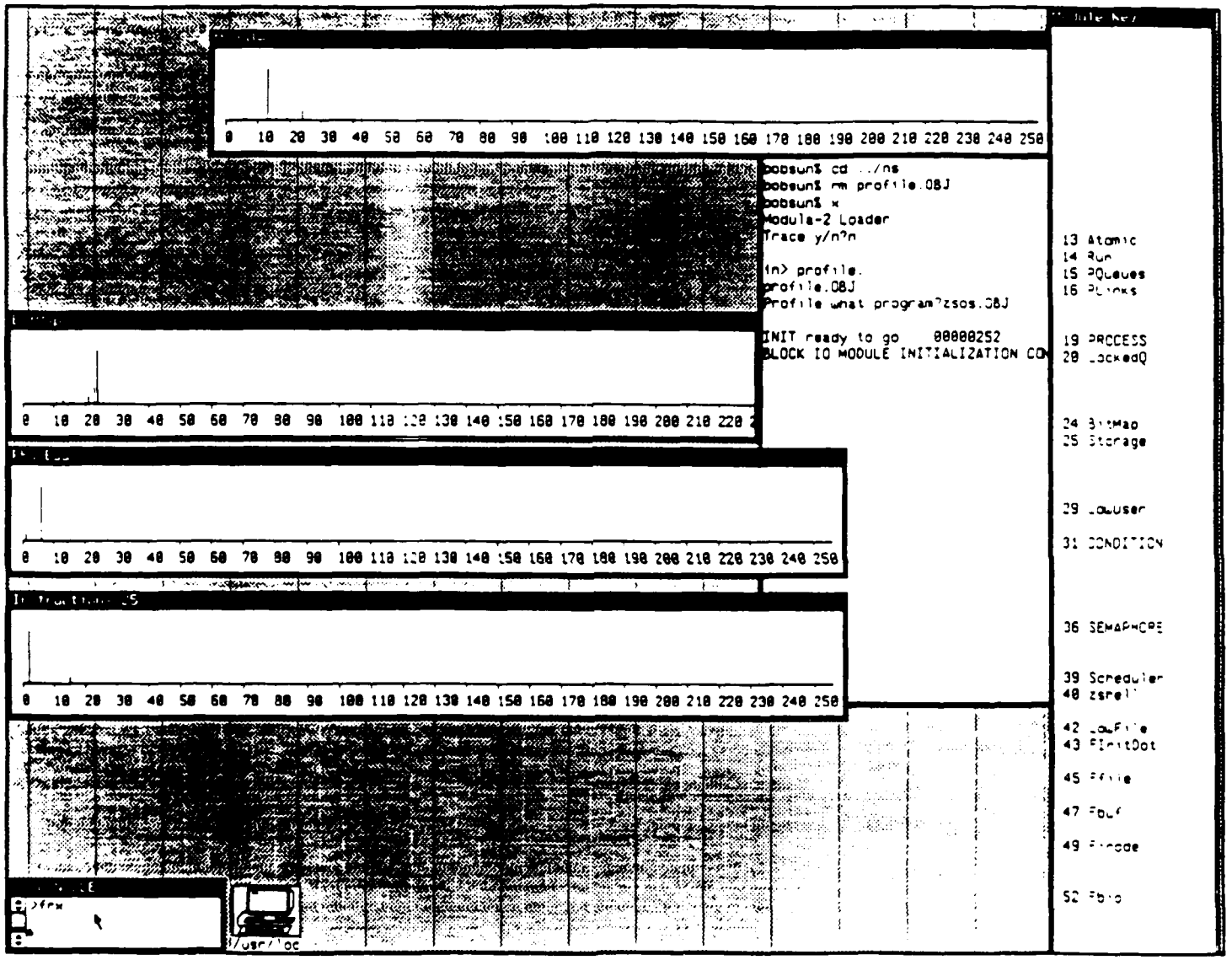

Figure 4. The StarLite Profiler

Efficient Locking for Process Control Operations

Gregory N. Fife
gnf3e@virginia.edu
uunet!virginia!uvacs!gnf3e
(804) 924-7605

Robert P. Cook
cook@uvacs.cs.virginia.edu
uunet!virginia!uvacs!cook
(804) 982-2215

Department of Computer Science
The University of Virginia
Charlottesville, Virginia 22903

The UNIX Operating System maintains a hierarchical family-tree ordering of the processes running on a machine. This tree is manipulated by three operations: fork, exit, and wait. The data structure that implements the process tree is a linear array; each entry or "slot" in the array contains the attributes of an individual process and the links that maintain the position of the process within the array. In the traditional implementations of UNIX, the process data structure can be accessed by at most one process at any time, and several of the operations that manipulate the tree run in time proportional to the number of children involved. We discuss an implementation of the UNIX semantics in which the process data structure may be locked one record at a time, allowing concurrent execution of the basic operations. We present a locking strategy that minimizes the locks acquired for any operation to proceed. We show that the tree operators can manipulate our data structure in constant time, and we show that deadlock can be efficiently avoided with our locking strategy.

1. Introduction

The UNIX operating system maintains a logical tree structure of processes in a kernel table. Clearly, some form of concurrency control is required to maintain the consistency of the data structure in the presence of multiple simultaneous operations. In a traditional UNIX implementation [Bach 1986][Leffler et al. 1989], a process running in the kernel cannot be pre-empted, and sequential access to the process table is guaranteed. In effect, a single lock governs the entire kernel state, including the process table. The inability to pre-empt an unimportant process in the kernel on behalf of a higher priority process makes this implementation less suitable for real-time applications. Also, exclusive access to the process table does not scale well to a large, possibly distributed system. If the total number of processes waiting to perform operations on the process data structure grows without bound, then the delay that each process incurs waiting to gain access to the structure will grow without bound as well. A finer granularity of locking is required.

In this paper, we consider locking the process data structure at the granularity of a single process record. In the remainder of this section, we discuss the semantics of the tree operations and the traditional UNIX implementation of these semantics. In Section 2, we present our own data structures and the constant time algorithms which operate upon them. In Section 3, we discuss the locking strategy that supports concurrent access to the data structure, and in Section 4, we present an efficient method for avoiding deadlocks caused by the method of Section 3. We conclude in Section 5.

The UNIX process model is illustrated in Figure 1. Each process is associated with a unique positive integer, or PID, which serves as a name for the process. In our diagram, the root process running "init" has PID 1, and leaf node executing "a.out" has PID 41. For each process, UNIX records PPID, or Parent Process ID. We show PPID's in parentheses.

The operation that performs process creation is called *fork*. When a process performs a fork, then a child process is created. The child has a new and unique PID, and the child is given a PPID equal to the PID of the caller of the fork. The new process is initialized as a copy of its parent. For example, in Figure 2, the shell, "sh," with PID 35 has performed a fork. The new process has PID 77 and PPID 35, and it is also currently running "sh".

The operations that remove a process from the tree are exit and wait. Exit causes a process to cease running; it may be initiated by the subject of the exit itself, or it may be induced from elsewhere in the system. In UNIX, an exiting process is marked as a *zombie*, and it remains in the process table until it is deallocated by a later cleanup

operation. Figure 3 shows the results of an exit by Process 77.

In the normal operation of UNIX, the cleanup of a zombie occurs when the parent performs a *wait* operation. The caller of wait searches among its children for a zombie. If one is found, then an exit status code and some accounting statistics are read from the process record, and the slot is then unlinked from the tree and deallocated. If a zombie is not found immediately, the parent sleeps until one is available. Figure 4 shows the system tree after the parent of Process 77 performs a wait.

The parent, of course, can defer waiting for an arbitrarily long period of time and leave the zombie in the system. The process slot held by the zombie cannot be immediately reallocated. The UNIX System V solution [Bach 1986] is to generate a "Death of Child" signal whenever an exit occurs. If the recipient of the signal has explicitly indicated to the kernel that it will ignore that signal, then the zombie is immediately deallocated by the kernel. We present a solution in Section 2 that solves the same problem without requiring specific actions from a user process.

Another concern with exit is the disposition of children of the deceased process. In UNIX, these *orphans* are made children of the init process. Figure 5 shows the reassignment of Process 41 after Process 35 performs an exit. Because the parent fields in the slots of each orphan are updated immediately to reflect the new position within the tree, the a UNIX exit which creates n orphans must run in O(n) time.

The process tree is embedded in a binary tree formed by pointers within the process table slots. For each process, the identity of a first child and a next sibling are recorded. Back links are also maintained; each process is associated with a previous sibling and, of course, a parent. Links that would represent the family tree of Figure 1 are shown in Figure 6.

2. The Data Structure

Having presented the UNIX operating system process operations, we now present the data structures that our scheme uses to implement them. We assume that there is a "User Record" associated with each process. No assumption is made about the arrangement of the records; they may be in a fixed-length array, or a dynamically allocated heap. A name or handle exists which can be used to reference each record; for the purposes of discussion, we call this name a `pUser`. We require that a lock be associated with each User Record, and we assume that a process will block until a requested lock is available. Further requirements for our locking mechanism are stated below. We assume a mechanism for binding PID's to User Records, and we assume an allocation and release

mechanism for unused records. We require from the latter a guarantee that no two forking processes will be granted the same User Record and that released User Records are not allocated for other purposes. Finally, we assume a mechanism, such as a semaphore, to allow a waiting process to block without holding any locks until a child has terminated.

Figure 6 shows the fields of the User Record which are relevant to our discussion. Following the UNIX implementation discussed earlier, we use the `Parent`, `FirstChild`, `NextSibling`, and `PrevSibling` fields to record the position of a record within the process tree. The remaining fields are based on the Phoenix Multiprocessor Operating System under development at the University of Virginia.

The `ResultHead` and `ResultTail` fields point to a linked queue of process return codes. When a process exits, it places its return code and PID in its parent's list and frees the User Record that it occupied. The Phoenix result list allows the immediate reuse of a zombie's User Record without requiring that a process take the explicit step of ignoring a "Death of Child" signal. The wait call now runs in O(1) time since it removes a record from the head of a result list, after possibly waiting for a wakeup action.

The `Generation` and `ParGeneration` fields allow individual exit operations to manipulate the process data structure in constant time, even if an arbitrary number of orphans must be handled. Specifically, the `Generation` field contains a unique integer value for the process currently bound to the User Record. The `ParGeneration` field is merely the generation value of the process's parent. When any process exits, the `Generation` field of the User Record that it had occupied is immediately updated. If the exiting process happens to be a parent, then the surviving children can determine that they are orphans at the time that they exit. A similar test can be used for any other operation which requires a correct PPID for a process. The work of resolving n orphans is spread out over the n exit calls which must yield exit codes to the root process, and each operation manipulates the tree in O(1) time.

The finite number of values which the `Generation` field can take on could potentially cause a problem with wraparound. However, if a 64-bit unsigned value is used, then the 2×10^{19} numbers available should be adequate for most applications.

3. The Locking Protocol

In this section, we consider the locks that the exit, fork, and wait operations must acquire in order to guarantee correct concurrent manipulation of the process data structure. We also consider interference from other system operations. At the time that a fork, exit, or wait is being executed, another process might take an action which involves some of the same User Records. The *kill* system call has this property; a process can call kill to send a signal to another process at any time. Therefore, we consider kill (and any other arbitrary operation) along with fork, exit and wait.

Kill. Concurrent operations on the general fields of a User Record are clearly undesirable. Therefore, we require that an arbitrary operation on a process acquire the lock on the target User Record before the operation is executed. This lock will also form the basis for preventing interference among kill and fork, wait, or exit.

Exit. A process that is being removed from the tree should clearly not be signaled. Therefore, an exiting process should acquire a lock on itself. If two siblings attempt to exit at the same time, then the linked list that they remove themselves from may be left in an inconsistent state. This can be avoided by requiring an exiting process to gain a lock on its parent before it proceeds.

Wait. A process might try to pull a result from its result list at the same time that a child tries to add a new result. Therefore, a process should acquire the lock on its own User Record before performing a wait.

Fork. A forking process must not add a child to its linked list at the same time that a child already there attempts to remove itself. If a process in a fork call acquires the lock on its own User Record, then we know that this cannot happen. If the process being created is assigned a PID as the last step of the fork operation, then another process cannot name the new process until the fork is completed. Thus, an arbitrary call such as kill cannot collide at the newly allocated User Record, and the overhead of locking the new record can be avoided.

Other Conflicts. In this locking scheme, a kill cannot interfere with an exit because all the records involved are locked. Kill cannot interfere with wait for the same reason, and the interactions between kill and fork are discussed above. A wait only involves the result list of the process making the call, and collisions with any other possible user of the data structure are covered by the locking rules given above. All collisions with exit have been dealt with. Collisions with fork involving the parent are prevented by the lock which the parent acquires on itself, and collisions involving the child are prevented by not giving the child a name until the fork is complete. Thus, the

locking scheme presented above maintains the consistency of the data structure in the face of all possible conflicts.

For example, suppose our Process 35 decides to exit. The locks it will acquire on itself and on its parent are shown with squares on Figure 8. Process 35 is protected against arbitrary interference by the lock it holds on its own record. Process 41 cannot begin an exit until Process 35 completes; therefore it will determine that it has become an orphan. The parent of Process 35 cannot access its result list with a wait, and it cannot access its sibling list with a fork. Finally, note that Process 61, the sibling of 35, cannot remove itself from the sibling list until 35 completes and releases the lock on the shared parent.

4. Deadlock

We now consider deadlock. We prevent deadlock by specifying an order in which all operations will attempt to acquire locks. For wait, fork, and kill, only one lock is required. For exit, we must acquire the lock on the process making the call first, and then on its original parent, it is still alive, and finally, on the User Record of the init process if necessary.

This rule can lead to deadlock. Suppose that an orphan and a child of the orphan exit concurrently, and suppose that each gains the lock on its own User Record. The child of the orphan will now try to lock the orphan's User Record, and if the child happens to occupy the User Record of the original parent, then the orphan will attempt to lock the User Record of its child. Because these deadlocks only occur when a process attempts to lock a record that it doesn't really need to lock anyway, the locking mechanism can detect the condition, abort the incorrect request, and unblock the orphan. To do this, the locking mechanism should be given the expected generation value of the parent record being locked. The locking mechanism should atomically examine the generation field of the target record and decide whether to abort or process the lock request. Also, whenever a process changes a generation field, it should notify the locking mechanism. At this time, any pending lock requests for that User Record should be aborted. If a process has its request for a lock on its parent aborted for either of these reasons, then it knows it is an orphan and can proceed to lock the init process record.

This locking rule guarantees that if a user process is holding a lock on a record at some level of the family tree, then it can only block trying to acquire a lock on a record closer to the root of the tree. Therefore, no cycle can occur. The partial order implied by the family tree is strong enough to prevent deadlock.

5. Conclusion

In this paper, we have shown how to implement the UNIX process management primitives for a large number of processes in an efficient manner. The data structure allows basic operations to run in constant time, and only a few locks need be acquired to allow operations to run in parallel. Manipulations of our linked lists do not require a lock on every record that is read or modified, and our exit operation requests at most three locks. Fork, wait, and kill can run with only one lock each. Because we lock processes and parents of processes, but never children of processes or siblings of processes, deadlock is easily avoided.

6. References

M. J. Bach, *The Design of the UNIX Operating System*, Englewood Cliffs: Prentice-Hall, 1986.

S. J. Leffler, M. K. McKusick, M. J. Karels, J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System* , Reading: Addison-Wesley, 1989.

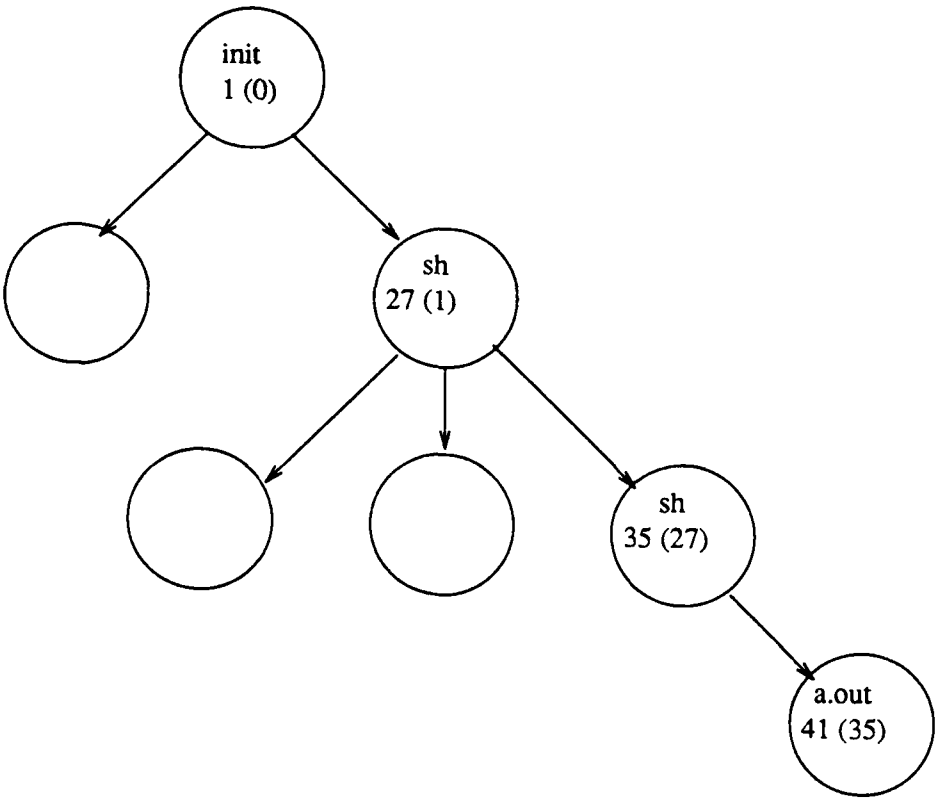

Figure 1
A family tree

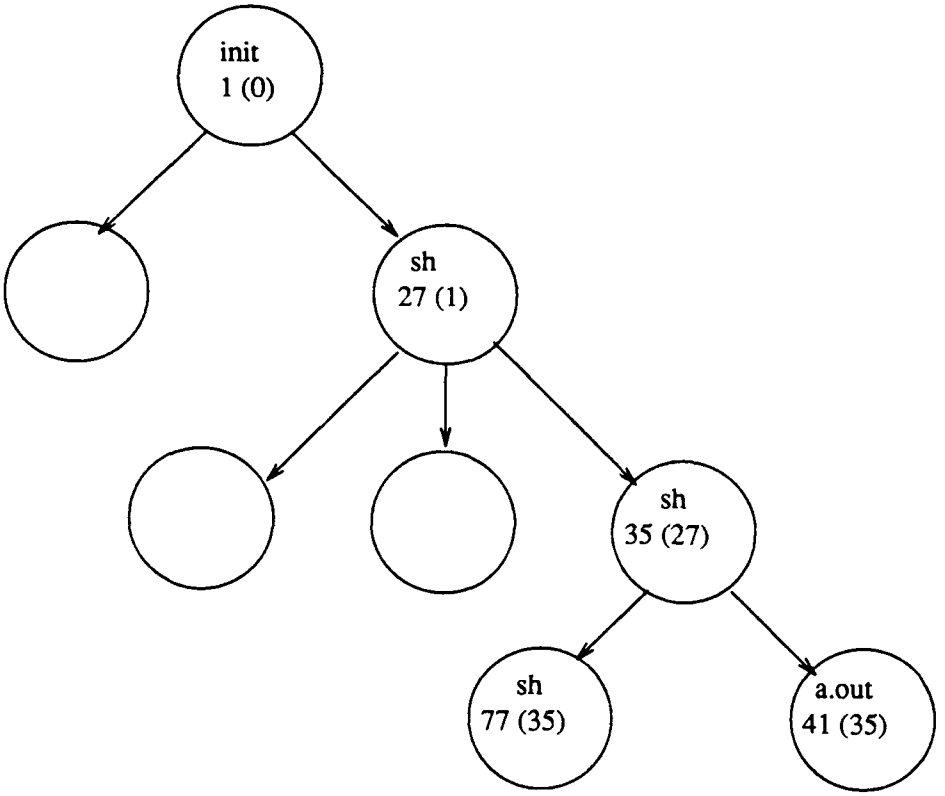

Figure 2
After a fork

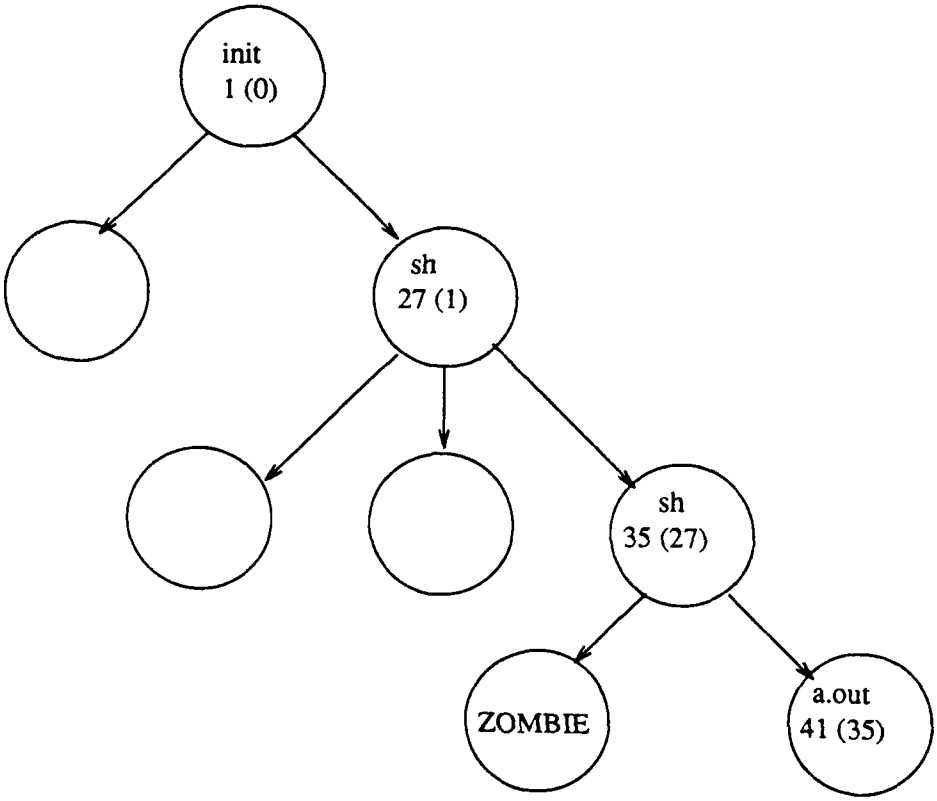

Figure 3
After an exit

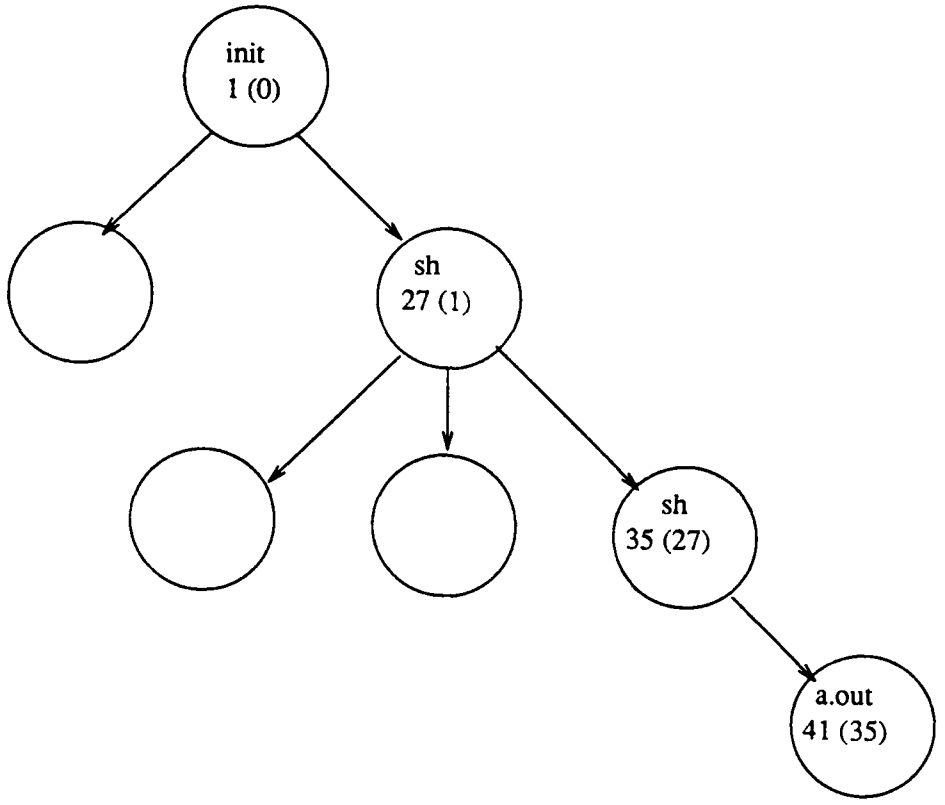

Figure 4
After a wait

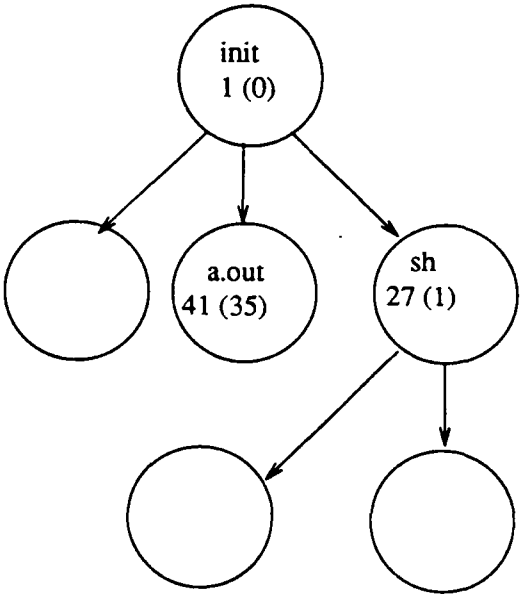

Figure 5
An Orphan

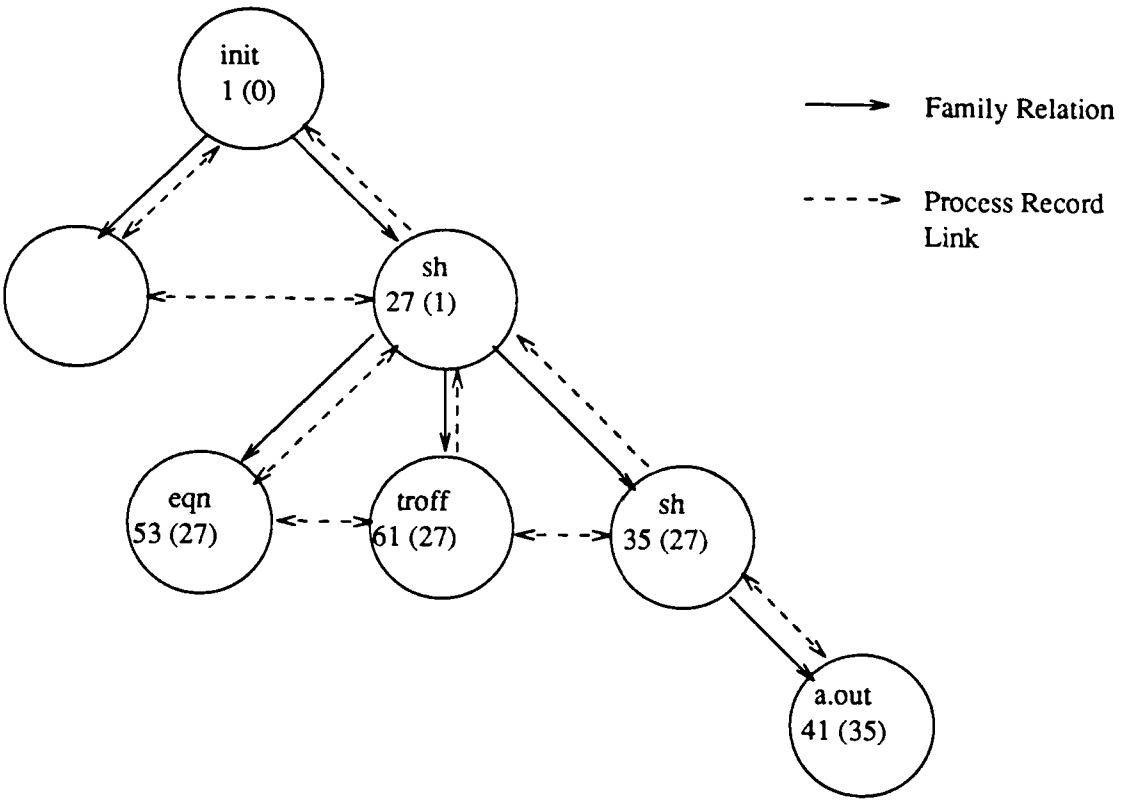

Figure 6
The Process Links

```
TYPE
  pUser ;    (* Index, Pointer, or Name to reference a User RECORD *)
  VLONGINT ;                          (* sufficiently long integer *)
  pRes   = POINTER TO Result ;              (* head of result list *)
  User   = RECORD
             Parent : pUser ;               (* parent of this process *)
             FirstChild : pUser ;             (* head of child list *)
             NextSibling : pUser ;        (* next entry in child list *)
             PrevSibling : pUser ;        (* prev entry in child list *)
             ResultHead : pRes ;      (* head of pending result list *)
             ResultTail : pRes ;      (* tail of pending result list *)
             Generation : VLONGINT ;(* generation # of this process *)
             ParGeneration : VLONGINT ;     (* generation of parent *)

             (* rest of user record of kernel process table follows *)

           END ;
```

Figure 7

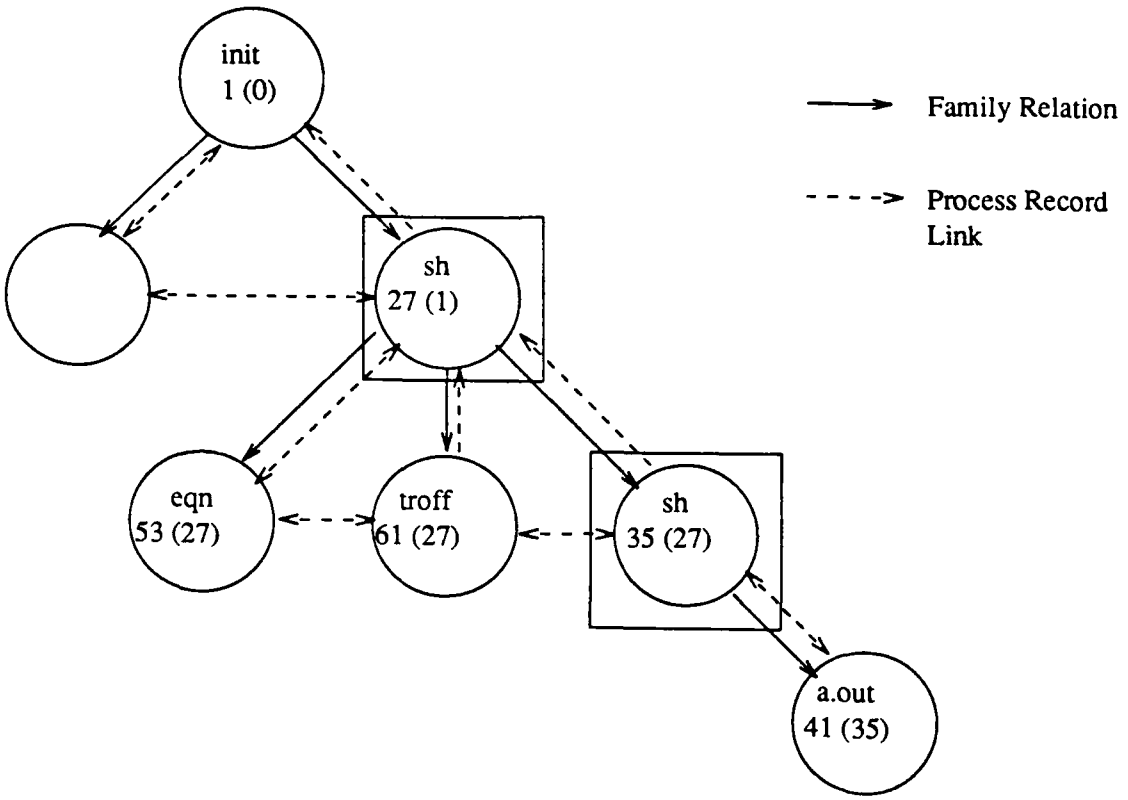

Figure 8
Exit Locking

RDB,
An Open, Real-Time, Relational Database Kernel

Robert P. Cook* and Sang H. Son
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

1.0 Introduction

In this paper, we discuss the attributes of RDB, which is an "open", real-time, relational database kernel. RDB was implemented using the StarLite[1] software development environment. It was inspired by the SDB system[2] created by Betz and Smith.

RDB is intended for use in embedded systems with requirements for high-performance and real-time priority and predictability guarantees. RDB is a tool that can be used to achieve these goals but it is the user's responsibility to use it properly. For example, RDB is completely reentrant and can be preempted in one context switch time to perform an action for a high priority process. Thus, a query can be interrupted for an update action and then restarted. However, if the low priority process holds locks that the high priority process needs (priority inversion), it is the user's responsibility to resolve the difficulty.

RDB is an "open"[3] system. It is implemented as a hierarchy of modules that are structured so that they can be easily modified or replaced. Furthermore, RDB does not depend on any operating system services. As a result, it is possible to manipulate ROM or memory-resident databases without having to depend on the access methods traditionally provided by operating systems. As a final point, RDB uses upcalls[4] to implement late binding of query and I/O operations.

By providing the user of RDB fine-grained control over its operation, it is simple to select implementation strategies that achieve performance and predictability goals. This can be contrasted with traditional database systems that operate as closed boxes, often with poor performance and predictability characteristics.

The following sections describe the relational model supported by RDB and a simple example that illustrates its use.

2.0 The RDB Model

The RDB kernel supports the following abstract data types: Schema, Relation, Attribute, Cursor, SortKey, SortList, SortLists, Selection, and Expression. Figure 1 illustrates the relationships among the various types.

A Schema in RDB describes the tuple format in terms of the position and type of the Attribute fields. At present, the only field types are text and numeric. The schema is disjoint from the data composing a Relation in order to provide options for real-time systems that are not normally found in traditional database systems. For example, it is possible to define a Relation's content as a file,

*This work is supported by ARO under contract DAAL03-87-K0090 and by ONR under contract N00014-86-K0245.

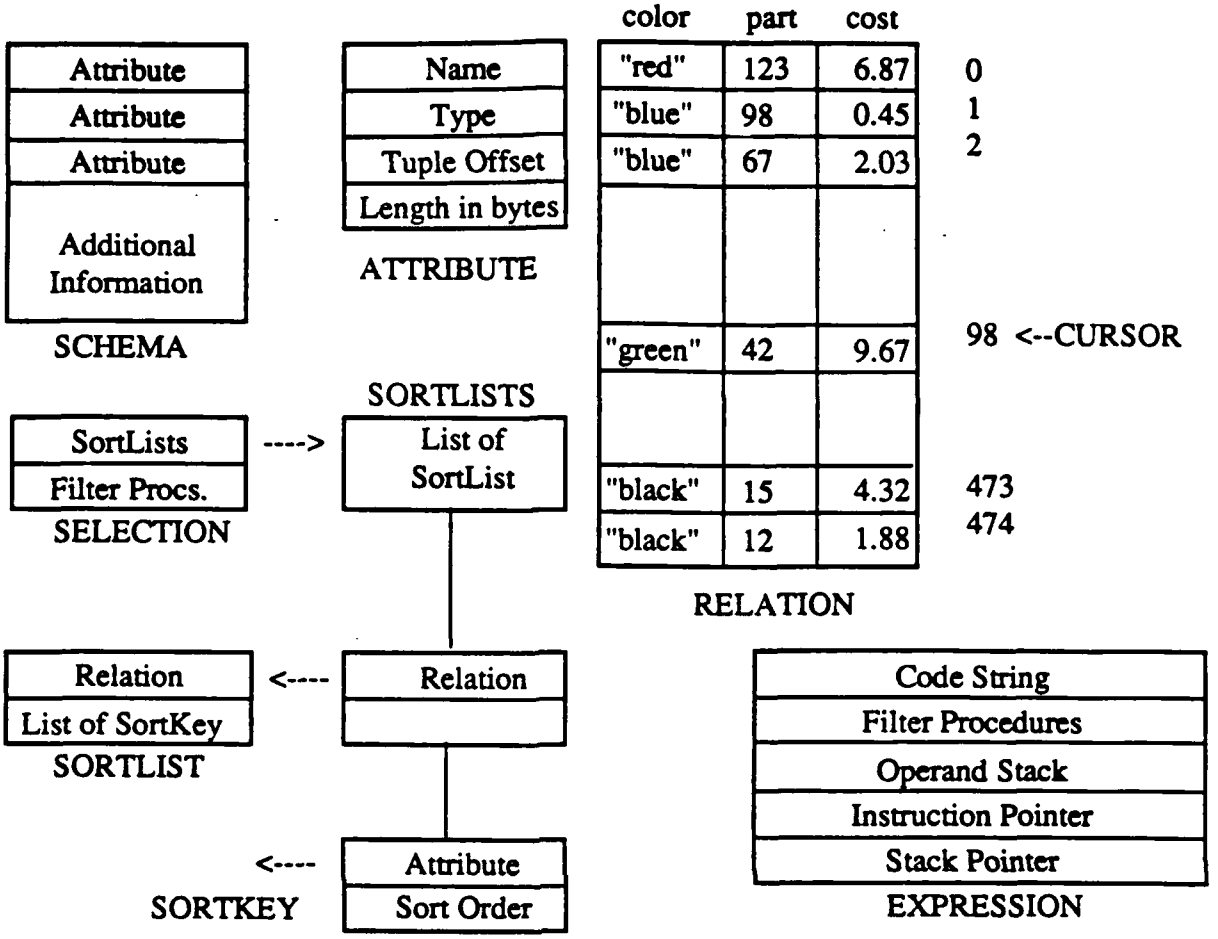

Figure 1. The RDB DataBase Model

but is is also possible to define derived relations. That is, the tuples making up a relation can be computed as requested or they can be generated from a data stream of sensor inputs[5]. Figure 1 illustrates a relation composed from three attributes: color (text), part number (numeric), and cost (numeric). Each Attribute specifies the field name and type as well as its position in the tuple and its length in bytes. The file consists of 475 tuples.

Once a schema has been defined and a relation selected, any number of Cursors can be opened. A cursor identifies a tuple in a relation. Cursors are used to "mark" the positions at which I/O operations are to occur in a database. In RDB, the actual I/O is performed using upcalls. An upcall is an invocation of a procedure variable that is bound after RDB is loaded but before a relation is accessed. The procedures to be invoked are specified when a relation is "connected" to the system for I/O.

Upcalls give the system the flexibility to implement ROM or memory-resident databases, derived or computed relations, and relations based on data streams. In essence, each relation can have a set of access procedures that are tuned to meet system performance and predictability goals. The system provides several traditional access methods, which the user is free to modify or to augment.

The SortKey is a central data structure in most database implementations. It defines the order relation to be used for an attribute when it is accessed. For example, the "part number" attrib-

ute in Figure 1 is sorted in descending numeric order.

A SortList identifies a relation and a list of sort keys. The keys represent the projection of the relation over which a particular operator is to be applied. If an attribute is referenced through a secondary index, the projection can sometimes be loaded by referring to the index rather than reading the tuples of the original relation. For a real-time system, the update costs associated with a secondary index must be compared with its efficiency advantages for query processing.

A SortLists is a list of Sortlist elements, where each SortList is itself a list of SortKeys. The SortLists data type represents the list of projections of relations that participate in multi-relation operations, such as a join.

The join operation is implemented by selection based on one or more expressions. A Selection data structure contains the input SortLists (relations and keys) as well as the upcall procedure variables that are used to filter the tuples in the input relations. Filtering can be applied during selection either when each tuple of a relation is input or when one tuple in each relation has been input.

Tuple filtering can be combined with expression filtering to achieve results that are not possible in a traditional database system. For example, any tuple filter has the ability to terminate selection. As a result, a query that cannot be completed by its original deadline can return a partial, or less accurate result, and still meet its timing constraints.

The Expression data type is implemented in a fashion that makes it orthogonal to the rest of the database kernel. It operates on code strings such as "e0000 s04blue =", which compares field zero in relation zero to the string "blue". If they are equal, the top of the operand stack is set to the Boolean value TRUE.

As with several other components of RDB, the Expression module uses an upcall procedure to bind the interpretatation of the "e" operator (load external). The user typically provides an Expression with a procedure that will return attribute values when presented with the arguments to "load external". However, the Expression module does not know that it is being used by a database system.

As a result, the user of RDB is free to generate the operands of an expression in a manner that is application dependent. For example, a traditional database system would lock out updates to a relation while a query was in progress. Locking can result in priority inversion which is an anathema in real-time systems.

With RDB, the selection filters and expression processing can be specified such that "compensation" is possible. That is, the updates are made to the relation and are simultaneously factored into the query so that neither the query process nor the update process are delayed. In a similar fashion, if records are deleted, the effect may be "subtracted" from a query in progress.

3.0 An RDB Example

The following example illustrates the use of RDB and the Phoenix real-time operating system, which is also part of StarLite, to implement a cyclic process that prints a "parts" report once every hour starting at a particular hour.

Phoenix provides an operator that transforms a procedure into a lightweight thread. Other operators allow a thread to set or change its priority and to delay until a selected time has arrived. Even though a delay operator for a relative amount of time has the same expressive power, we have found that using an absolute time specification results in programs that are more likely to perform as the user intended. This is particularly true for very fine-grained timing control operations.

```
PROCEDURE reportGenerator(initHr:CARDINAL);
BEGIN
```

```
  SetPriority(Self(), 7);
  LOOP
    AtSecond.At(initHr*60);
    IF initHr =24 THEN initHr := 1;
    ELSE initHr := initHr+1;
    END;
    print();
  END;
END reportGenerator;
```

Figure 2. A Cyclic Report Generator

```
PROCEDURE print();
  VAR r : Relation;
      s : SortList;
      input : SortLists;
      sel : Selection;
      e : Expression;
BEGIN
  r := RFind ("partsFile");
  ROpenSort(r, s);
  RAddKey(s, "cost", "ascending");
  RAddKey(s, "color", "");
  ROpenSortLists(input);
  RAddSortList(input, s);
  sel := ROpenSelect(input, e, FIONULL, printr);
  RInterpret.ROpen(e, "e0000 s04blue =", f, sel);
  RSelect(sel);
  RInterpret.Close(e);
  RCloseSelect(sel);
  RCloseSortLists(input);
END print;
```

Figure 3. Initiate Record Selection

In Figure 3, the "print" procedure associates a Relation variable with the database file and schema. Next, the SortLists is constructed to describe the projections of the input relations that are to be printed (in this case just one relation). Notice that the "cost" and "color" keys are permuted from the storage order. In general, a SortList can permute the keys for both the input and output relations.

After the SortLists is initialized, the Selection and Expression data structures are initialized. One of the arguments used to create the "sel" variable is the upcall procedure "printr" that actually produces the report. One of the arguments (procedure "f") to create an expression is an upcall procedure that converts attributes in the input tuples into expression operands. Figure 4 presents an outline of "f" and "printr".

```
PROCEDURE f(relation, attr:CARDINAL
          arg:ADDRESS; VAR (*out*) o:Operand);
  (* set Operand to field "attr" in "relation" *)
  VAR s : Selection;
    pT : pTuple;
    pSK : pSortKey;
BEGIN
  s := arg;   (* remembered by Expression *)
  pSK := RGetSKey(s, relation, attr, pT (*out*) );
  o.pT := pT;
  o.offset := pSK^.offset;
  o.length := pSK^.length;
END f;

PROCEDURE printr(s:Selection; arg:ADDRESS);
  (* filter items to be printed in the report *)
  VAR e : Expression;
    pT : pTuple;
    pO : pOperand;
BEGIN
  e := arg;  (* remembered during selection *)
  pO := RInterpret.Evaluate(e);
  IF NOT pO^.b THEN RETURN; END;
  (* FOR EACH SortList in s DO
    pT := RGetSBuf(s, i);
    Write the selected attribures in pT^
  *)
  InOut.WriteLn;  (* terminate output line *)
END printr;
```

Figure 4. Upcall Procedures

When one tuple has been read for each relation selected as input, the "printr" procedure is invoked. This procedure in turn evaluates an expression to select the tuples to be printed in the report.

Whenever the expression evaluator encounters the "Load External", "e" operator, it performs an upcall to the "f" procedure that was passed as an argument to ROpen to create an Expression variable. One of the arguments to "f" is the address of an operand descriptor. It is the procedure's responsibility to map the relation and attribute indices to a SortKey. The SortKey is

then used to retrieve an attribute value, which is passed back to the evaluator as an operand.

When the evaluator completes, it returns an operand descriptor for the value on the top of the operand stack. For the "printr" procedure, the result is a Boolean value. If it is true, the fields are printed in the report. If it is false, the fields are ignored and selection continues.

RDB implements a number of very flexible options for expression evaluation that space does not permit us to describe. For real-time systems, the two most important are expression preemption and contextual reevaluation.

In the former, any expression can be preempted at any time by more critical expressions or other system actions. Using contextual reevaluation, it is possible for a query to modify its expression while it is being evaluated to return a less accurate result in order to meet timing constraints.

4.0 Summary

The RDB database kernel is intended for use in stand-alone applications that have "hard" timing requirements. It is an "open" system in that the user can manipulate interfaces not normally available in traditional database systems to "tune" performance to application requirements. The use of upcalls also adds great flexibility to both selection and expression processing options.

RDB does not currently support transactions, locking, or recovery. It can, however, operate on either local relations or remote files by using RPC. We are implementing additional functionality as part of a layered design for database operations. The layers are implemented so that the end-user can add or subtract features to meet the performance or timing requirements of embedded systems.

References

[1] Cook, R.P., StarLite, A Visual Simulation Package for Software Prototyping, **Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments**, (Dec. 1986) 102-110, also **SIGPLAN Notices 22**, 1(Jan. 1987).

[2] Betz, D. and D.N. Smith, SDB - A Simple Database System, from documentation provided by Pat Watson at IBM Manassas (Nov. 1988).

[3] Lampson, B.W. and R.F. Sproull, An Open Operating System for a Single-User Machine, **Proc. of the 7th Symposium on Operating System Principles**, (Dec. 1979) 98-105.

[4] Clark, D., The Structuring of Systems Using Upcalls, **Proc. of the 10th Symposium on Operating System Principles**, (Dec. 1985) 171-180.

[5] Snodgrass, R., A Relational Approach to Monitoring Complex Systems, **ACM Transactions on Computer Systems 6**, 2(May 1988) 157-196.

DISTRIBUTION LIST

1 - 50	U. S. Army Research Office P. O. Box 12211 4300 S. Miami Boulevard Research Triangle Park, NC 27709-2211 Attention: Dr. David W. Hislop Electronics Division
51	Dr. David W. Hislop Electronics Division U. S. Army Research Office P. O. Box 12211 4300 S. Miami Boulevard Research Triangle Park, NC 27709-2211
52*	Office of Naval Research Resident Representative 2135 Wisconsin Avenue, N. W. Suite 102 Washington, DC 20007 Attention: Mr. Michael McCracken Administrative Contracting Officer
53 - 54	R. P. Cook, CS
55	A. K. Jones, CS
56 - 57	E. H. Pancake, Clark Hall
58	SEAS Preaward Administration Files

*Cover Letter only

JO#2916:ph